SYBASE SQL Server™
System Administration Guide

SYBASE SQL Server Release 10.0

Document ID: 32500-01-1000-03

Change Level: 3

Last Revised: June 17, 1994

Principal authorship: Server Publications Group

Document ID: 32500-01-1000

This publication pertains to SYBASE SQL Server Release 10.0 of the SYBASE database management software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of the agreement.

Document-Back Guarantee

Sybase welcomes corrections and comments on its documents. If you mark typographical errors, formatting errors, errors of fact, or areas that need clarification in any Sybase user's manual and send copies of marked-up pages to us, we will send you a clean copy of the manual, absolutely free**.**

Send pages to the Publications Operations Department at the address below. Please include your Site ID number.

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

(510) 922-3500
Fax (510) 922-5340

Document Orders

Customers may purchase additional copies of any document or the right to make photocopies of documentation for their in-house use.

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

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

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

Restricted Rights Legend

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

Table of Contents

2. Roles in SQL Server

3. Managing Physical Resources

4. Managing SQL Server Logins and Database Users

5. Managing User Permissions

6. Checking Database Consistency

7. Developing a Backup and Recovery Plan

8. Backing Up and Restoring User Databases

9. Backing Up and Restoring the System Databases

10. Managing Free Space with Thresholds

11. Diagnosing System Problems

12. Fine-Tuning Performance and Operations

13. Locking

14. Managing Multiprocessor Servers

15. Managing Remote Servers

16. Auditing

17. Language, Sort Order, and Character Set Issues

18. Converting Character Sets Between SQL Server and Client

A. Reserved Words

B. The System Tables

C. The *pubs2* Database

D. Glossary

Index

List of Figures

List of Tables

Preface

This manual, the *SYBASE SQL Server™ System Administration Guide*, describes how to administer and control SQL Server™ databases independent of any specific database application.

Audience

This manual is intended for SYBASE SQL Server System Administrators and Database Owners.

How to Use this Book

This manual contains two sections. Section I, “Basic Concepts,” covers basic system administration issues:

- Chapter 1 describes the structure of the SYBASE® system.
- Chapter 2 describes different system administration roles and capabilities.
- Chapter 3 discusses the physical placement of databases, tables, and indexes, and the allocation of space to them.
- Chapter 4 covers the addition and management of database users.
- Chapter 5 describes SQL Server’s protection system and the privileges that different users have and can be granted.
- Chapter 6 describes how to use the database consistency checker, **dbcc**, to detect and fix database problems.
- Chapter 7 discusses the capabilities of the Backup Server, and how to develop your backup strategy.
- Chapter 8 discusses how to recover user databases.
- Chapter 9 discusses how to recover system databases.
- Chapter 10 discusses managing space with thresholds.

Section II, “Special Topics,” contains information needed for special environments or infrequently encountered situations:

- Chapter 11 discusses SQL Server and Backup Server error handling, and shows how to shut down servers and kill user processes.

- Chapter 12 describes how to configure SQL Server for the best performance.
- Chapter 13 describes how SQL Server locks database pages and objects, and presents strategies for reducing lock contention.
- Chapter 14 discusses system administration issues unique to symmetric multiprocessor environments.
- Chapter 15 covers management of remote servers and remote users.
- Chapter 16 describes auditing on SQL Server.
- Chapter 17 discusses international issues, such as the files included in the Language Modules and how to change SQL Server's language, sort order, or character set.
- Chapter 18 discusses character set conversion between SQL Server and clients in a heterogeneous environment.
- Appendix A, "Reserved Words", lists Transact-SQL reserved words, and includes a list of words that may be used as keywords in SQL 92 implementations.
- Appendix B, ''The System Tables'' lists each system table and the definition of all columns in each table. A pull-out entity relationship diagram is included at the back of the book.
- Appendix C, ''The pubs2 Database'' describes the *pubs2* database in detail. Also included is an entity relationship diagram of all the tables in *pubs2*.
- Appendix D is a glossary of terms used in this book that relate to system administration and database management issues.

Related Documents

Other manuals you may find useful are:

- *SQL Server Reference Manual*, which contains detailed information on all of the commands and system procedures discussed in this manual.
- *SQL Server Utility Programs*, which documents the SYBASE utility programs such as **isql** and **bcp** which are executed from the operating system level.
- *System Administration Guide Supplement*, which documents operating-system-specific system administration tasks.

- *Transact-SQL User's Guide*, which documents Transact-SQL®, Sybase's enhanced version of the relational database language. It serves as a textbook for beginning users of the SYBASE database management system.
- *SQL Server Installation Guide*, which describes the installation procedures for SQL Server.
- *Master Index for Server Publications*, which combines the indexes of the *SQL Server Reference Manual*, *Transact-SQL User's Guide*, and *System Administration Guide*. Use it to locate various topics in different contexts throughout the SYBASE documentation.
- *What's New in Sybase SQL Server Release 10.0?* , which describes the new features in Release 10.0.

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:

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.

Table 1: Syntax Statement Conventions

Key	Definition
[ ]	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.

Table 1: Syntax Statement Conventions (continued)

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

  sp_dropdevice [*device_name*]

  or, for a command with more options:

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

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

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

  ```
  select * from publishers
  ```

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

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

  (3 rows affected)
  ```

Case

You can disregard case when you type keywords:

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

Obligatory Options {You Must Choose At Least One}

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

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

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

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

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

- **One Item in Square Brackets:** You 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 (three dots) means that you can **repeat** the last unit as many times as you like. In this syntax statement, buy is a required keyword:

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

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

Expressions

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

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

Table 2: Types of Expressions Used in Syntax Statements

If You Need Help

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

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

Basic Concepts

1 Overview of System Administration

Introduction

Administering SQL Server databases includes tasks such as:

- Installing SQL Server and Backup Server
- Granting roles and permissions to SQL Server users
- Managing and monitoring the use of disk space, memory, and connections
- Backing up and restoring databases
- Diagnosing system problems
- Fine-tuning SQL Server to achieve the best performance

In addition, System Administrators may have a hand in certain database design tasks, such as enforcing integrity standards. This function may overlap with the work of application designers.

Although a System Administrator concentrates on tasks that are independent of the applications running on SQL Server, she or he is likely to be the person with the best overview of all the applications. For this reason, a System Administrator can advise application designers about the data that already exists on the SQL Server, make recommendations about standardizing data definitions across applications, and so on.

The *System Administration Guide* is concerned with physical storage issues, backup and recovery, fine-tuning SQL Server, and so on. Functions specific to an application—data definition and maintaining referential integrity—are covered in other manuals. Installation of SQL Server is covered in the *SYBASE SQL Server Installation Guide*.

However, the distinction between what is and what is not specific to an application is sometimes a bit fuzzy. Owners of user databases will consult certain sections of this book. Similarly, System Administrators and Database Owners will use the *Transact-SQL User's Guide* (especially the chapters on data definition, stored procedures, and triggers).

The roles of the System Administrator, the Database Owner, and other users of SQL Server are further discussed in Chapter 2, ‘‘Roles in SQL Server’’.

Roles Required for System Administration Tasks

Many of the commands and procedures discussed in this document require you to be a System Administrator or System Security Officer. These special SQL Server roles are discussed in Chapter 2, ‘‘Roles in SQL Server’’.

Other sections of this document are relevant to Database Owners. A Database Owner’s username within the database is “dbo.” You cannot log in as “dbo”: a Database Owner logs in under his or her SQL Server login name, and is recognized as “dbo” by SQL Server only while using his or her database.

Using *isql* to Perform System Administration Tasks

Nearly all of the system administration tasks described in this guide require you as a System Administrator to connect to SQL Server with the utility **isql.** For more information about **isql**, see the *SQL Server Utility Programs* manual.

The System Tables

The *master* database contains system tables that keep track of information about SQL Server as a whole. In addition, each database (including the *master* database) contains system tables that keep track of information specific to that database.

All of the SQL Server-supplied tables in the *master* database (SQL Server’s controlling database) are considered system tables. In addition, each user database is created with a subset of these system tables. The system tables may also be referred to as the data dictionary or the system catalogs.

A *master* database and its tables are created when you install SQL Server. The system tables in a user database are automatically created when the **create database** command is issued. The names of all system tables start with “sys”. An explanation of the system tables and their columns is included in Appendix B, ‘‘The System Tables’’. Some of the system tables are discussed in detail in later chapters.

Querying the System Tables

You can query the system tables just like any other tables. For example, here's a statement that returns the names of all the triggers in the database:

```
select name
from sysobjects
where type = "TR"
```

In addition, SQL Server supplies stored procedures (called system procedures), many of which provide shortcuts for querying the system tables.

These system procedures provide information from the system tables:

sp_commonkey	**sp_helpindex**	**sp_helpsort**
sp_configure	**sp_helpjoins**	**sp_helptext**
sp_dboption	**sp_helpkey**	**sp_helpthreshold**
sp_estspace	**sp_helplanguage**	**sp_helpuser**
sp_help	**sp_helplog**	**sp_lock**
sp_helpconstraint	**sp_helpremotelogin**	**sp_monitor**
sp_helpdb	**sp_helprotect**	**sp_spaceused**
sp_helpdevice	**sp_helpsegment**	**sp_who**
sp_helpgroup	**sp_helpserver**	

For complete information on the system procedures, see the *SQL Server Reference Manual*, Volume 2.

Changing Data in System Tables

SQL Server's system tables contain information critical to the operation of your databases. Data in these tables is inserted, updated, or deleted by Transact-SQL commands (such as **create** or **drop** commands) or by system procedures. Under ordinary circumstances, you do not need to perform direct data modifications to system tables.

Generally, you should update these tables only when:

- You are instructed to do so by Technical Support, or
- Instructions in the *SYBASE Troubleshooting Guide* or this guide instruct you to update a system table.

If you must update system tables, you must issue an **sp_configure** command that enables system table updates. While this command is in effect, any user with appropriate permission could modify a system table. Other guidelines for direct changes to system tables are:

- Modify system tables only inside a transaction. Issue a **begin transaction** command before you issue the data modification command.
- Check to see that only the rows you wished to change were affected by the command, and that the data was changed correctly.
- If the command was incorrect, issue a **rollback transaction** command. If the command was correct, use **commit transaction.**

◆ ***WARNING!***

Some system tables should not be altered by any user under any circumstances. Some system tables are built dynamically by system processes, contain encoded information, or display only a portion of their data when queried. Imprudent ad hoc updates to certain system tables can make SQL Server unable to run, make database objects inaccessible, scramble permissions on objects, or terminate your user session.

Keys in System Tables

Primary, foreign, and common keys for the system tables have been defined in the *master* and *model* databases (the database used as a template for creating user databases). A report on defined keys is available by executing the system procedure **sp_helpkey.** For a report on columns in two system tables that are likely join candidates, execute **sp_helpjoins**. The system tables diagram included at the end of this book shows the relationships between columns in the system tables.

System Procedures

The names of all the system procedures begin with **sp_**. They are located in the *sybsystemprocs* database, but many of them can be run from any database.

Except for system procedures that update only tables in *master*, if a system procedure is executed in a database other than *sybsystemprocs*, it operates on the system tables in the database from which it was executed. For example, if the Database Owner of *pubs2* runs **sp_adduser** from *pubs2*, the new user is added to *pubs2..sysusers*.

Permissions on the system procedures are discussed in Chapter 5, "Managing User Permissions", and in the *SQL Server Reference Manual*, Volume 2.

Using System Procedures

If a parameter value for a system procedure contains reserved words, punctuation, or embedded blanks, it must be enclosed in single or double quotes. If the parameter is an object name and the object name is qualified by a database name or owner name, the entire name must be enclosed in single or double quotes.

System procedures can be invoked by sessions using either chained or unchained transaction modes. However, the system procedures that modify data in *master's* system tables cannot be executed from within a transaction, since this could comprise recovery. The system procedures that create temporary work tables cannot be run from transactions.

If no transaction is active when you execute system procedures, SQL Server turns off chained mode and sets **transaction isolation level 1** for the duration of the procedure. Before returning, the session's chained mode and isolation level are reset to their original settings. For more information about transaction modes and isolation levels, see Volume 1 of the *SQL Server Reference Manual*.

All system procedures report a return status. For example:

```
return status = 0
```

means that the procedure executed successfully.

System Procedure Tables

The system procedures use several **system procedure tables** in the *master* database to convert internal system values (for example, status bits) into human-readable format. One of them, *spt_values*, is used by a wide variety of system procedures including **sp_configure**, **sp_dboption**, **sp_depends**, **sp_help**, **sp_helpdb**, **sp_helpdevice**, **sp_helpindex**, **sp_helpkey**, **sp_helprotect**, and **sp_lock**.

The table *spt_values* is only updated by upgrade; you should never modify it. To see how it is used, execute **sp_helptext** to look at the text for one of the system procedures that references it.

The other system procedure tables include *spt_monitor* and *spt_committab* and tables needed by the Catalog Procedures. In addition, several of the system procedures create and then drop temporary tables. For example, **sp_helpdb** creates *#spdbdesc*; **sp_helpdevice** creates *#spdevtab*; and **sp_helpindex** creates *#spindtab*.

Creating System Procedures

Many of the system procedures are explained in this manual, in sections to which they are relevant. The *SQL Server Reference Manual*, Volume 2 lists and describes all of them.

System Administrators can write system procedures that can be executed from any database. Simply create a stored procedure in *sybsystemprocs* and give it a name that begins with **sp_**. The *uid* of the stored procedure must be 1, the *uid* of the Database Owner.

Most of the system procedures that you create yourself will query the system tables. You can also create stored procedures that modify the system tables, although this is not recommended.

To create a stored procedure that modifies system tables, a System Security Officer must first turn on the **allow updates** configuration variable. Any stored procedure created while this variable is set on will **always** be able to update system tables, even when **allow updates** is turned off. Here's how you'd create a stored procedure that updates the system tables:

- Issue **sp_configure** and **reconfigure with override** so that direct updates to system tables are allowed (see Chapter 12, "Fine-Tuning Performance and Operations").
- Create the stored procedure with the **create procedure** command.
- Issue **sp_configure** and **reconfigure** again, to disallow direct updates to system tables.

◆ ***WARNING!***

Use extreme caution if you must modify system tables. Test procedures that modify system tables in development or test databases, not in your production database.

System Databases

When you install SQL Server, it has these system databases:

- The *master* database
- The *model* database
- The system procedure database, *sybsystemprocs*
- The temporary database, *tempdb*

You can optionally install:

- The auditing database, *sybsecurity*. Use **sybinit** to install this database.
- The sample database, *pubs2*, by using **isql** and the **installpubs2** script.
- The *sybsyntax* database. See the *System Administration Guide Supplement* for your platform for more information on *sybsyntax*.

The *master*, *model*, and temporary databases all reside on the device named during installation, which is known as *d_master*. The *master* database is contained entirely on the master device and cannot be expanded onto any other device. All other databases and user objects should be created on other devices.

The *sybsecurity* database should be installed on its own device and segment. (This is discussed in the *Secure SQL Server Installation Guide*.)

The *sybsystemprocs* database can be installed on a device of your choice. You may wish to modify the **installpubs2** script and the scripts that install the *sybsyntax* database to share the device you created for *sybsystemprocs*.

➤ ***Note***

The ***installpubs2*** script does not specify a device in its **create database** statement, so it will be created on the default device. At installation time, the master device is the default device. To change this you can edit the script or follow directions later in this document for adding more database devices and designating default devices.

master Database

The *master* database (*master*) controls the user databases and the operation of SQL Server as a whole. It keeps track of:

- User accounts (in *syslogins*)
- Remote user accounts (in *sysremotelogins*)
- Remote servers that this server can interact with (in *sysservers*)
- Ongoing processes (in *sysprocesses*)
- The configurable environment variables (in *sysconfigures*)
- System error messages (in *sysmessages*)
- The databases on SQL Server (in *sysdatabases*)
- The storage space allocated to each database (in *sysusages*)
- The tapes and disks mounted on the system (in *sysdevices*)
- The active locks (in *syslocks*)
- Character sets (in *syscharsets*) and languages (in *syslanguages*)
- Users who hold server-wide roles (in *sysloginroles*)
- Server roles (in *syssrvroles*)
- SQL Server engines online (in *sysengines*)

You must be in the *master* database in order to issue the **create** or **alter database**, **disk init**, **disk refit**, **disk reinit** and the disk mirroring commands.

Controlling Object Creation in master

When you first install SQL Server, only System Administrators can create objects in the *master* database, because they implicitly become "dbo" of any database they use. It is not a good idea to routinely add user objects to the *master* database device. Any objects created on the *master* database device should be used for the administration of the system as a whole. Permissions in *master* should remain set so that most users cannot create objects there.

Another way to discourage users from creating objects in *master* is to change users' default databases (the database to which the user is connected when he or she logs in) with the system procedure **sp_modifylogin**. This procedure is discussed in Chapter 4, "Managing SQL Server Logins and Database Users".

If you create system procedures, create them in the *sybsystemprocs* database, rather than in *master.*

Protecting Your Server by Backing up master

To be prepared for hardware or software failure on your server, the two most important housekeeping tasks are:

- Performing frequent backups of the *master* database (as well as all of your user databases).

➤ ***Note***

Back up the *master* database with **dump database** each time you create, alter or drop any device, database, or database object, and each time you execute a stored procedure that changes it. See Chapter 9, "Backing Up and Restoring the System Databases".)

- Keeping a copy (preferably off-line) of these system tables: *sysusages, sysdatabases, sysdevices, sysloginroles* and *syslogins*. You may want to create a script to perform these commands:

```
select * from sysusages order by vstart
select * from sysdatabases
select * from sysdevices
select * from sysloginroles
select * from syslogins
```

  If you have copies of these scripts, and a hard disk crash or other disaster makes your database unusable, you will be able to use the recovery procedures described in Chapter 9, "Backing Up and Restoring the System Databases". If you do not have current copies of this information, full recovery of your SQL Server becomes much more difficult after damage to the *master* database.

model Database

The *model* database is also supplied with SQL Server. It provides a template or prototype for new user databases. Each time the **create database** command is issued, SQL Server makes a copy of the *model* database, and then extends the new database to the size requested in the **create database** command.

➤ ***Note***

A new database can never be smaller than the *model* database.

The *model* database contains the system tables required for each user database. It can be modified to customize the structure of newly created databases—everything you do to *model* will be reflected in each new database. Here are some of the changes that are commonly made to *model*:

- Adding user-defined datatypes, rules, or defaults.
- Adding users who are to be given access to all databases on SQL Server.
- Granting default privileges, particularly for guest accounts
- Setting database options such as **select into/bulkcopy.** The setting of options will be reflected in all new databases. Their original value in *model* is "off". Database options are described in Chapter 12, "Fine-Tuning Performance and Operations".

Typically, most users are not granted permission to modify the *model* database. There isn't much point in granting read permission either, since its entire contents are copied into each new user database.

The size of *model* cannot be larger than the size of *tempdb*. SQL Server displays an error message if you try to increase the size of *model* without making *tempdb* at least as large.

➤ ***Note***

Keep a backup copy of the *model* database and back up *model* with **dump database** each time you change it. In case of media failure, restore *model* as you would a user database.

sybsystemprocs Database

SYBASE system procedures are stored in the database *sybsystemprocs.* When a user in any database executes any stored procedure that starts with the characters "sp_", SQL Server first looks for that procedure in user's current database. If there is no procedure there with that name, SQL Server looks for it in *sybsystemprocs*. If there's no procedure in *sybsystemprocs* by that name, SQL Server looks for the procedure in *master.*

If the procedure modifies system tables (for example, **sp_adduser** modifies the *sysusers* table*)*, the changes are made in the database from which the procedure was executed.

To change the default permissions on system procedures, you must modify these permissions in *sybsystemprocs*.

➤ ***Note***

If you make changes to *sybsystemprocs* or add your own stored procedures to the database, you should backup the database regularly.

tempdb Database

SQL Server has a temporary database, *tempdb*. It provides a storage area for temporary tables and other temporary working storage needs (for example, intermediate results of **group by** and **order by**). The space in *tempdb* is shared among all users of all databases on the server.

Create temporary tables either by preceding the table name in a **create table** statement with a pound sign (#), or by specifying the name prefix "*tempdb*..".

Temporary tables created with a pound sign are accessible only by the current SQL Server session: users on other sessions cannot access them. These non-sharable temporary tables are destroyed at the end of the current session. The first 13 bytes of the table's name, including the pound sign (#), must be unique. SQL Server assigns the names of such tables a 17-byte number suffix. (You can see the suffix when you query *tempdb..sysobjects*.)

Temporary tables created with the "*tempdb*.." prefix are stored in *tempdb* and can be shared among SQL Server sessions. SQL Server does not change the names of temporary tables created this way. The table exists either until you reboot SQL Server, or until its owner drops it using **drop table**.

System procedures (for example, **sp_help**) work on temporary tables, but only if you use them from *tempdb*.

If a stored procedure creates temporary tables, the tables are dropped when the procedure exits. Temporary tables can also be dropped explicitly before a session ends.

Each time you reboot SQL Server, it copies *model* to *tempdb,* clearing the database. Temporary tables are not recoverable.

The default size of *tempdb* is 2 megabytes. Certain activities may make it necessary to increase the size of *tempdb*. The most common of these are:

- Large temporary tables.
- A lot of activity on temporary tables (which fills up *tempdb*'s logs).
- Large sorts, or many simultaneous sorts. Subqueries and aggregates with **group by** also cause some activity in *tempdb*.

You can increase the size of *tempdb* with the **alter database** command. *tempdb* is initially created on the master device. Additional space can be added on the master device or on any other database device.

No special permissions are required to use *tempdb*—that is, to create temporary tables or to execute commands that may require storage space in the temporary database.

sybsecurity Database

The *sybsecurity* database contains the audit system for SQL Server. It consists of:

- The *sysaudits* table, which contains the audit trail. All audit records are written into *sysaudits.*
- The *sysauditoptions* table, which contains rows describing the global audit options.
- All of the other default system tables, derived from *model*.

The audit system is discussed in Chapter 16, "Auditing".

pubs2 Database

Installing the sample database (*pubs2*) is optional. Provided as a learning tool, *pubs2* is the basis of most of the examples in the SQL Server document set. The sample database is illustrated in Appendix C, "The pubs2 Database".

Using the Sample Database

Many of the examples in the SQL Server documentation set are based on *pubs2*. See the *System Administration Guide Supplement* for your platform for information about installing *pubs2*.

The sample database contains a guest user mechanism that allows any authorized SQL Server user to access it. The guest user has been given a fairly wide range of privileges in *pubs2*, including permission to **select**, **insert**, **update**, and **delete** all the user tables. For more on the guest user mechanism, and a list of the guest permissions in *pubs2*, see Chapter 4, ‘‘Managing SQL Server Logins and Database Users’’.

If possible, each new user should be given a clean copy of *pubs2* so that she or he won't become confused by changes made by other users. *pubs2* requires 2 megabytes. If you want it to be created on a specific database device, edit the *pubs2* database installation script.

If space is a problem, you could instruct users to issue the **begin transaction** command before updating the sample database. After the user is finished updating, he or she can issue the **rollback transaction** command in order to undo the changes.

sybsyntax Database

The syntax database, *sybsyntax*, contains the syntax of commands and language libraries for SYBASE products. Users can retrieve this information using the system procedure **sp_syntax.** There are separate installation scripts for different SYBASE products.

See the *System Administration Guide Supplement* for your platform for information about installing this database.

2 Roles in SQL Server

Introduction

In SQL Server users can be granted special operational and administrative roles. Roles provide individual accountability for users performing system administration and security-related tasks. Roles are granted to individual server login accounts, and actions performed by these users can be audited and attributed to them.

These users gain their special status by being granted their roles with the **sp_role** command. These roles are:

- The System Administrator
- The System Security Officer
- The Operator

In addition, there are two kinds of object owners, who gain special status because of the objects they own. These ownership types are:

- The owner of a user database
- The owner of database objects

All of these are discussed in the following sections.

System and Security Administration Roles

The System Administrator, System Security Officer, and Operator roles are essential for managing SQL Server.

These roles are granted to individual users with the **sp_role** command. See "Granting and Revoking Roles" on page 2-6 for more information.

More than one login account on a SQL Server can be granted any role, and one account can possess more than one role.

The System Administrator

A System Administrator performs administrative tasks unrelated to specific applications. The System Administrator is not necessarily one individual; the role can be granted to any number of individual login accounts. In a large organization, the System Administrator's role may be carried out by several people or groups. It is important, however, that the System Administrator's functions be centralized or very well coordinated.

System Administrator Tasks

System Administrator tasks include:

- Installing SQL Server
- Managing disk storage
- Granting permissions to SQL Server users
- Transferring bulk data between SQL Server and other software programs
- Modifying, dropping, and locking server login accounts
- Monitoring SQL Server's automatic recovery procedure
- Diagnosing system problems and reporting them as appropriate
- Fine-tuning SQL Server by changing the configurable system parameters
- Creating user databases and granting ownership of them
- Granting and revoking the System Administrator role
- Setting up groups (which are convenient in granting and revoking permissions)

System Administrator Permissions

A System Administrator operates outside the object and command protection system—SQL Server does no permission checking when this user accesses objects. A System Administrator takes on the identity of Database Owner in any database he or she enters, including *master*, by assuming the user ID 1.

There are several commands and system procedures that only a System Administrator can issue, and on which permissions cannot be transferred to other users. For detailed information on System Administrator permissions, see Chapter 5, "Managing User Permissions".

System Security Officer

The System Security Officer is responsible for security-sensitive tasks in SQL Server, such as:

- Creating server login accounts

- Granting and revoking the System Security Officer and Operator roles
- Changing the password of any account
- Setting the password expiration interval
- Managing the audit system

The System Security Officer has no special permissions on database objects, except in the *sysaudits* database where only a System Security Officer can access the *sysaudits* table. There are also several system procedures that only a System Security Officer can execute and on which permissions cannot be transferred to other users.

The Operator

An operator is a user who can back up and load databases on a server-wide basis. The Operator role allows a single user to use the **dump database, dump transaction**, **load database**, and **load transaction** commands to back up and restore all databases on a server without having to be the owner of each one. These operations can be performed in a single database by the Database Owner and the System Administrator.

Data Ownership Roles

There are two types of owners recognized by SQL Server:

- Database Owners
- Database Object Owners

Database Owner

The Database Owner is the creator of a database or someone to whom database ownership has been transferred. The System Administrator grants users the authority to create databases with the **grant** command.

A Database Owner logs into SQL Server using his or her assigned login name and password. In other databases, that owner is known by his or her regular user name; in his or her own database SQL Server recognizes the user as "dbo." When SQL Server is installed, the "sa" login is the Database Owner of the *master* database.

Database Owner Tasks

The owner of a database may:

- Run the system procedure **sp_adduser** to allow other SQL Server users access to the database
- **grant** other users permission to create objects and execute commands within the database

These tasks are discussed in Chapter 4, “Managing SQL Server Logins and Database Users”.

Database Owner Permissions

The Database Owner has full permissions on objects inside the database that he or she owns.

Database Object Owner

Database objects are tables, indexes, views, defaults, triggers, rules constraints and procedures. A user who creates a database object is its owner. The Database Owner must first grant the user permission to create the particular type of object. There are no special login names or passwords for database object owners.

Database Object Owner Tasks

The database object owner creates an object using the appropriate **create** statement, and then grants permission to other users.

Database Object Owner Permissions

The creator of a database object is automatically granted all permissions on it. System Administrators also have all permissions on the object, as long as they qualify the object name with the owner’s name (for example, *fred.table2*). The owner of an object must explicitly grant permissions to other users before they can access it. Even the Database Owner cannot use an object directly unless the object owner grants him or her the appropriate permission. However, the Database Owner can always use the **setuser** command to impersonate any other user in the database.

➤ ***Note***

When a database object is owned by someone other than the Database Owner, the user (including a System Administrator) must qualify the name of that object with the object owner's name—*ownername.objectname*—to access the object. If an object or a procedure needs to be accessed by a large number of users, particularly in ad hoc queries, having these objects owned by "dbo" greatly simplifies access.

Managing Roles in SQL Server

When it is installed, SQL Server includes an account called "sa" that is automatically granted the System Administrator, System Security Officer, and Operator roles during installation. The "sa" user is the Database Owner of the system databases. There are two ways to go from here:

- You can use the "sa" account to perform all system administration and security-related functions. Any user who knows the "sa" password can use the account, and any actions performed can only be traced to "sa"; it is not possible to determine the individual user who was logged in as "sa".
- To increase accountability:
  - After installing SQL Server, use the "sa" account to create new server logins for users who are to be granted the System Administrator and System Security Officer roles.
  - Still logged in as "sa", grant the correct roles to these users using **sp_role**.
  - A user who has been granted the System Security Officer role can then log into his or her own account, lock the "sa" account (using **sp_locklogin**), and create logins for other server users (using **sp_addlogin**).

➤ ***Note***

If you decide to lock the "sa" account, be sure to check all scripts that may contain the "sa" login name and password. These can include scripts that perform backups, run **bcp**, or perform **dbcc** checking. The scripts cannot run if they are meant to run as "sa" and that account is locked. Change the logins in those scripts to the name of a user with the correct role.

Granting and Revoking Roles

The System Security Officer can grant the System Security Officer and Operator roles to other users, and the System Administrator can grant the System Administrator role. This is done with the **sp_role** system procedure. The syntax is:

```
sp_role {"grant" | "revoke"},
   {sa_role | sso_role | oper_role}, login_name
```

grant and **revoke** specify whether you are granting or revoking the role.

The roles are **sa_role**, **sso_role**, or **oper_role**. Only one role can be specified in each execution of **sp_role.**

login_name is the name of the user to whom the role is being granted or revoked.

For example, this command:

```
sp_role "grant", sa_role, arthur
```

grants the role of System Administrator to "arthur".

When you grant a role to a user, it takes effect the next time the user logs into SQL Server. However, the user can immediately enable the role by using the **set role** command. For example, this command:

```
set role "sa_role" on
```

immediately enables the System Administrator role for the user.

You cannot revoke a role from a user while the user is logged in.

You cannot lock or drop the last remaining System Security Officer's or System Administrator's account. The system procedures **sp_droplogin**, **sp_locklogin**, and **sp_role** ensure that there is always at least one unlocked login that possesses the System Security Officer role, and one that possesses the System Administrator role.

Turning Your Roles On and Off

When you log into SQL Server, all roles granted to you are automatically enabled. Use the **role** option of the **set** command to turn any of your roles off or back on again for your current session. The syntax is:

```
set role "{sa_role | sso_role | oper_role}" {on | off}
```

This can be useful if, for example, you have been granted the System Administrator role, which means that you assume the identity of

Database Owner within any database that you use. If you wish to assume your “real” user identity, execute this command:

```
set role "sa_role" off
```

If you are granted a role during a session and wish to activate it immediately, use **set role** to turn it on.

Checking for Roles

The following sections describe commands and procedures that allow you to check roles granted to yourself or others.

Displaying Login Account Information

You can use the **sp_displaylogin** system procedure to display information about a login account. The syntax is:

sp_displaylogin [*login_name*]

If you are not a System Security Officer or System Administrator, you can get information only about your own account and you do not need to use the *login_name* parameter. **sp_displaylogin** displays your server user ID, login name, full name, roles, date of last password change, and whether your account is locked.

If you are a System Security Officer or System Administrator, you can use the *login_name* parameter to access information about any login.

The show_role Function

The **show_role** function displays any roles that are currently enabled for your login session. The syntax is:

```
select show_role()
```

It returns NULL if you have no roles enabled.

Checking for Roles in Stored Procedures: the proc_role Function

Use **proc_role** within a stored procedure to guarantee that only users with a specific role can execute it. While **grant execute** can also restrict execute permission on a stored procedure, users without the required role might inadvertently be granted permission to execute it. Only **proc_role** provides a fail-safe way to prevent inappropriate access to a particular stored procedure.

proc_role takes a string for the required role (**sso_role, sa_role,** or **oper_role**) and returns 1 if the invoker possesses it. Otherwise, it returns 0.

For example, here is a procedure that uses **proc_role** to see if the user has the **sa_role** role:

```
create proc test_proc
as
if (proc_role("sa_role") = 0)
begin
    print "You don't have the right role"
    return -1
end
else
    print "You have SA role"
    return 0
```

3 Managing Physical Resources

Introduction

SQL Server is designed to make reasonable default decisions about many aspects of storage management—where databases, tables, and indexes are placed, and how much space is allocated for each of them. However, the System Administrator has control over the physical placement of databases, tables, indexes, and the allocation of space to them.

In addition, after an application has stabilized and its data-handling requirements have been determined, SQL Server provides ways to refine storage management. These refinements are not an essential aspect of data definition, but a means of improving performance.

This chapter covers these aspects of storage management:

- Issues in storage management decisions
- The system tables that manage storage
- Initializing database devices
- Designating default database devices
- Disk mirroring for nonstop recovery in case of physical disk crashes
- Creating user databases with the **create database** command
- Putting the transaction log on a separate device with the **log on** clause to **create database**
- Enlarging user databases with the **alter database** command
- Removing databases with the **drop database** command
- Creating and using segments
- Finding information on storage

Responsibility for storage allocation and management is often centralized. In most installations, the System Administrator retains complete control over these matters.

Commands for Managing Resources

Table 3-1 illustrates the major commands a System Administrator uses to manage the allocation of physical resources.

Command	Task
disk init **name = *"dev_name"*** **physname = *"phys_name"*** **etc.**	Makes a physical device available to a particular SQL Server. Assigns a database device name (*dev_name*) used to identify the device in other SQL Server commands.
disk mirror **name = *"dev_name"*** **mirror = "*phys_name*"** **etc.**	Mirrors a database device on a specific physical device.
sp_diskdefault *"dev_name"* **etc.**	Adds *dev_name* to the general pool of default database space.
create database ... on *dev_name* or **alter database ... on *dev_name***	Makes database devices available to a particular SQL Server database. The **log on** clause to **create database** places the database's logs on a particular database device.
create database ... or **alter database ...**	When used without the **on *dev_name*** clause, these commands allocate space on the default database devices.
sp_addsegment *seg_name,* ***dbname, devname*** and **sp_extendsegment *seg_name,*** ***dbname, devname***	Creates a segment, named collection of space, from the devices available to a particular database.
create table ... on *seg_name* or **create index ... on *seg_name***	Creates database objects, placing them on a specific segment of the database's assigned disk space.
create table ... or **create index ...**	When used without **on *seg_name***, tables and indexes occupy the general pool of space allocated to the database (the default devices).

Table 3-1: Commands for Allocating Physical Resources

Considerations in Storage Management Decisions

The System Administrator of a SQL Server installation must make many decisions regarding the physical allocation of space to SQL Server databases. The major considerations in these choices are:

- **Recovery:** Disk mirroring and / or maintaining logs on a separate physical device provide two mechanisms for full recovery in the event of physical disk crashes.
- **Performance**: For certain tables or databases where speed of disk reads and writes is crucial, properly assigning database objects to physical devices yields performance improvements. Disk mirroring will slow the speed of disk writes.

Recovery

Recovery is the key motivation for using several disk devices. Non-stop recovery can be accomplished by mirroring database devices. Full recovery can also be ensured by storing a database's log on a separate physical device.

Keeping Logs on a Separate Device

Unless a database device is mirrored, full recovery in case of media failure requires that a database's transaction log be stored on a different device from the actual data (including indexes) of a database. You can use the **log on** clause of **create database** to guarantee that log records will be stored on a separate device. In the event of a hard disk crash, an up-to-date database can be recreated by loading a dump of the database, and then applying the log records which were safely stored on another device.

Mirroring

Non-stop recovery in the event of a hard disk crash is guaranteed through the mechanism of mirroring SQL Server devices to a separate physical disk. Mirroring the database devices containing the actual data and indexes (not just the device containing the transaction log) is required for recovery without downtime.

Performance

System performance can be improved by placing logs and database objects on separate devices:

- Placing a table on one hard disk and non-clustered indexes on another ensures that physical reads and writes are faster, since the work is split between two disk drives.
- Splitting large tables across two disks can improve performance, particularly for multi-user applications.

Status and Defaults at Installation Time

Installing SQL Server is covered in the *SQL Server Installation Guide*. The installation program and scripts initialize the master device and set up the *master, model, sybsystemprocs, sybsecurity,* and temporary databases for you.

When you first install SQL Server:

- The *master, model,* and *tempdb* databases are installed on the master device *d_master.*
- The *sybsystemprocs* database is installed on a device that you choose at installation time.
- Three segments are created in each database: *system*, *default*, and *logsegment*.
- The default storage device for all user-created databases is *d_master.*
- If you have chosen to install the audit database, *sybsecurity,* it is located on its own device.

The System Tables that Manage Storage

Two system tables in the *master* database and two more in each user database track the placement of databases, tables (including the transaction log table, *syslogs*), and indexes. The relationship between the tables is illustrated in Figure 3-1.

The *sysdevices* Table

The *sysdevices* table in the *master* database contains one row for each database device and may contain a row for each dump device (tape, disk, or operating system file) available to SQL Server.

The **disk init** command adds entries for database devices to *master..sysdevices*. Dump devices, added with the system procedure

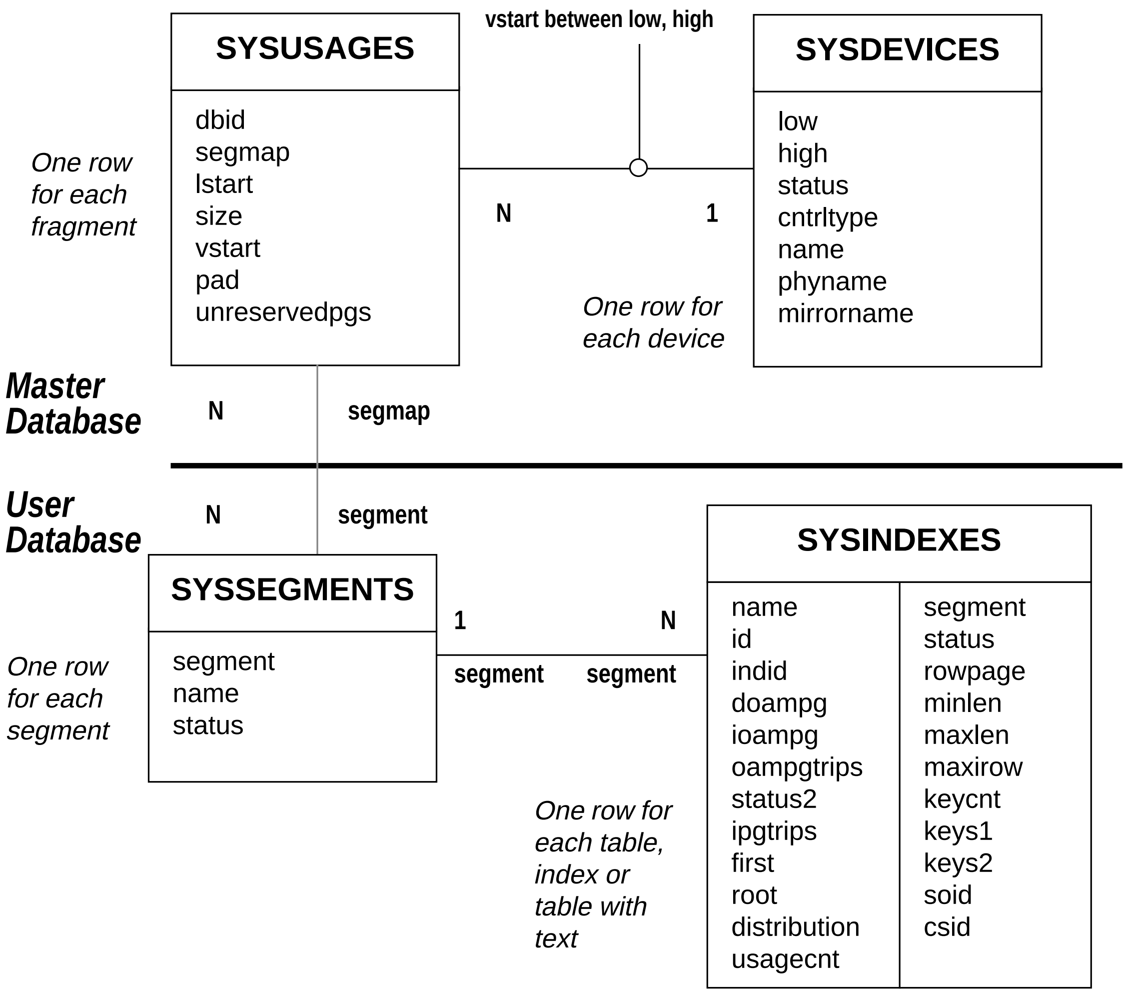


Figure 3-1: The System Tables that Manage Storage

sp_addumpdevice, are discussed in Chapter 7, ''Developing a Backup and Recovery Plan''.

sysdevices stores two names for each device:

- A "logical name" or "device name" to be used in all subsequent storage-management commands, stored in the *name* column of *sysdevices*. This is usually a user-friendly name, perhaps indicating the planned use for the device, for example "logdev" or "userdbdev".
- The "physical name", the actual operating system name of the device. You only use this name in the **disk init** command; after that, all SQL Server data storage commands use the logical name.

You place a database or transaction log on one or more devices by specifying the logical name of the device in **create database** or **alter database** statements. The **log on** clause to **create database** places a database's transaction log on a separate device (a must to ensure full recoverability). The log device must also have an entry in *sysdevices* before you can use **log on**.

A database can reside on one or more devices, and a device can store one or more databases.

The *sysusages* Table

The *sysusages* table in the *master* database keeps track of all of the space that you assign to all SQL Server databases.

create database and **alter database** allocate new space to the database by adding a row to *sysusages* for each database device or device fragment. When you allocate only a portion of the space on a device with **create** or **alter database**, that portion is called a **fragment**.

The system procedures **sp_addsegment, sp_dropsegment**, and **sp_extendsegment** change the *segmap* column in *sysusages* for the device that is mapped or unmapped to a segment.

The *syssegments* Table

The *syssegments* table, one in each database, lists the segments in a database. A segment is a collection of the database devices and/or fragments available to a particular database. Tables and indexes can be assigned to a particular segment, and thereby to a particular physical device, or can span a set of physical devices.

create database makes default entries in *syssegments*. The system procedures **sp_addsegment** and **sp_dropsegment** add and remove entries from *syssegments*.

The *sysindexes* Table

The *sysindexes* table lists each table and index, and the segment where each table, clustered index, nonclustered index, and each chain of text pages is stored.

New rows are created in *sysindexes* by **create table** and **create index** commands.

Initializing Database Devices

A database device stores the objects that make up databases. The term **device** does not necessarily refer to a distinct physical device: it can refer to any piece of disk (such as a partition) or a file in the file system used to store databases and their objects.

Each database device or file must be prepared and made known to SQL Server before it can be used for database storage. This process is called **initialization**.

Once a database device is initialized, it can be:

- Allocated to the pool of space available to a user database
- Allocated to a user database, and assigned to store a specific database object or objects
- Used to store a database's transaction logs
- Designated as a default device for **create** and **alter database** commands

A System Administrator initializes new database devices with the **disk init** command. **disk init** does the following:

- Maps the specified physical disk device or operating system file to a "database device" name
- Lists the new device in *master..sysdevices*
- Prepares the device for database storage

➤ ***Note***

Before you run **disk init**, see the *System Administration Guide Supplement* for your platform for information about choosing a database device and preparing it for use with SQL Server. You may wish to re-partition the disks on your computer to provide maximum performance for your SYBASE databases.

disk init divides the database devices into **allocation units** of 256 2K pages, a total of a half megabyte (on Stratus, 256 4K pages, one megabyte). In each 256-page allocation unit, the **disk init** command initializes the first page as the allocation page, which will contain information about the database (if any) that resides on the allocation unit.

◆ ***WARNING!***

After you run the disk init command, be sure to use dump database to dump the *master* database. This makes recovery easier and safer in case *master* is damaged. If you add a device and fail to back up *master*, you may be able to recover the changes with disk reinit. See Chapter 9, "Backing Up and Restoring the System Databases".

disk init Syntax

Here's the syntax of the **disk init** command:

```
disk init
   name = "device_name" ,
   physname = "physicalname" ,
   vdevno = virtual_device_number ,
   size = number_of_blocks
   [, vstart = virtual_address ,
       cntrltype = controller_number]
   [, contiguous] (OpenVMS only)
```

disk init Examples

On UNIX:

```
disk init
  name = "user_disk",
  physname = "/dev/rxy1a",
  vdevno = 2, size = 5120
```

On OpenVMS:

```
disk init
 name = “user_disk”,
 physname = “disk$rose_1:[dbs]user.dbs”,
 vdevno = 2, size = 5120,
 contiguous
```

Specifying a Logical Device Name with disk init

The *device_name* must be a valid identifier. This name is used in the **create database** and **alter database** commands, and in the system procedures that manage segments. The logical device name is known only to SQL Server, not to the computer system on which the server runs.

Specifying a Physical Device Name with disk init

The *physicalname* of the database device gives the name of a raw disk partition (UNIX) or foreign device (OpenVMS), or the name of an operating system file. On PC platforms you can use only operating system filenames for the *physicalname*.

Choosing a Device Number for disk init

vdevno is an identifying number for the database device. It must be unique among devices used by SQL Server. Device number 0 represents the device named *d_master*, that stores the system catalogs. Legal numbers are between 1 and 255, but the highest number must be one less than the number of database devices for which your system is configured. For example, for a system with the default configuration of 10 devices, the legal device numbers are 1 through 9. To see the configuration value for your system, execute **sp_configure devices**, without giving a second parameter, and check the *run_value*, the last column in the output:

```
sp_configure devices
```

```
name            minimum  maximum  config_value run_value
--------------- -------- -------- ------------ ---------
devices                4      256            0        10
```

To see the numbers already in use for **vdevno**, look in the *device_number* column of the report from **sp_helpdevice**, or, the following query lists all the device numbers currently in use:

```
select distinct low/16777216
    from sysdevices
    order by low
```

SQL Server is originally configured for 10 devices. You may be limited to a smaller number of devices by operating system constraints. See the discussion of **sp_configure**, which is used to change configuration options, in Chapter 12, ‘‘Fine-Tuning Performance and Operations’’.

If you issue a **disk init** that does not succeed for any reason, and you need to re-issue the **disk init** command, you must either use a different value for **vdevno**, or you must restart SQL Server.

Specifying the Device Size with disk init

The **size** of the database device must be given in 2K blocks (4K on Stratus). There are 512 2K blocks (256 4K blocks) in a megabyte.

If you are planning to use the new device for the creation of a new database, the minimum **size** is the larger of:

- The size of *model*. When you install SQL Server, *model* uses 2 megabytes, 1024 2K blocks. Use **sp_helpdb model** to see the current size of *model*.
- The configuration value **database size.** Use **sp_configure** and look at the *run_value* for **database size.**

If you are initializing a database device for a transaction log or for storing small tables or indexes on a segment, the **size** can be as small as 512 2K blocks (1 megabyte).

If you are initializing a raw device (UNIX) or foreign device (OpenVMS), determine the size of the device from your operating system as described in the *System Administration Guide Supplement* for your platform. Use the maximum size available; once you have initialized the disk for use by SQL Server, you cannot use any space on the disk for other purposes.

disk init uses **size** to compute the value for the high virtual page number in *sysdevices.high*.

Optional Parameters for disk init

vstart is the starting virtual address, or the offset in 2K blocks, for SQL Server to begin using the database device. The default value (and usually the preferred value) of **vstart** is zero.

The optional **cntrltype** keyword specifies the disk controller. Its default value is zero. Reset it only if instructed to do so.

contiguous, an option for OpenVMS systems only, forces the database file to be created contiguously.

➤ ***Note***

To perform disk initialization, the user who started SQL Server must have the appropriate operating system permissions on the device that is being initialized. (This does not apply to SQL Servers running on PCs.)

Getting Information about Devices

The system procedure **sp_helpdevice** provides information about the devices in the *sysdevices* table.

When used without a device name, **sp_helpdevice** lists all the devices available on SQL Server. When used with a device name, it lists information about that device. Here **sp_helpdevice** is used to report information on the master device:

```
sp_helpdevice master
```

```
device_name physical_name description
----------- ---------------------------------------------------------
master      d_master      special, default disk, physical disk, 20 MB

status      cntrltype     device_number     low      high
------      ----------    --------------    ------   -------
3           0             0                 0        9999
```

Each row in *master..sysdevices* describes:

- A dump device (tape, disk, or file) to be used for backing up databases, or
- A database device to be used for database storage.

The initial contents of *sysdevices* are operating-system dependent. Entries in *sysdevices* usually include:

- One for *d_master.*
- One for the *sybsystemprocs* database, which you can use to store additional databases such as *pubs2* and *sybsyntax*, or for user databases and logs.
- Two for tape dump devices.

If you installed auditing, there will also be a separate device for *sybsecurity.*

The *low* and *high* fields represent the page numbers that have been assigned to the device. For dump devices, they represent the media capacity of the device.

The *status* field in *sysdevices* is a bitmap which indicates the type of device, whether a disk device will be used as a default storage device when users issue a **create** or **alter database** command without specifying a database device, and disk mirroring information. Here are the bits and their meanings:

Bit	Meaning
1	default disk, may be used by any **create** or **alter database** command that doesn't specify a location
2	physical disk
4	logical disk (not used)

Table 3-2: Status Bits in sysdevices

Bit	Meaning
8	skip header
16	dump device
32	serial writes
64	device mirrored
128	reads mirrored
256	secondary mirror side only
512	mirror enabled
2048	used internally; set after **disk unmirror, side = retain**

Table 3-2: Status Bits in sysdevices (continued)

Dropping Devices

To drop database and dump devices, use the system procedure **sp_dropdevice**. The syntax is:

```
sp_dropdevice device_name
```

You cannot drop a device that is in use by a database. You must drop the database first.

sp_dropdevice removes the device name from *sysdevices*. **sp_dropdevice** does not remove an operating system file that is being dropped as a database device: it only makes the file inaccessible to SQL Server. You must use operating system commands to delete a file after using **sp_dropdevice**.

You must restart SQL Server after you drop a device because the server has a process that is accessing the dropped device; there is no other way to kill the process. Restarting frees up the virtual device number. Stop the server with **shutdown**; restart it with **startserver.** See the *SQL Server Utility Programs* manual for information on the **startserver** utility.

If a **disk init** fails for any reason, you may also need to restart SQL Server to free up the virtual device number.

Designating Default Devices

To create a pool of default database devices to be used by all SQL Server users for creating databases, use the **sp_diskdefault** command after the devices are initialized with **disk init**. **sp_diskdefault** marks these devices in *sysdevices* as default devices. Whenever users create databases (or alter databases) without specifying a database device, new disk space is allocated from the pool of default disk space.

The syntax for **sp_diskdefault** is:

```
sp_diskdefault device_name, {defaulton | defaultoff}
```

You are most likely to use the **defaultoff** option in order to remove the master device from the pool of default space:

```
sp_diskdefault master, defaultoff
```

The following command makes *sprocdev*, the device that holds the *sybsystemprocs* database, a default device:

```
sp_diskdefault sprocdev, defaulton
```

A SQL Server can have multiple default devices. They are used in the order in which they appear in the *sysdevices* table (i.e., alphabetical order). When the first default device is filled, the second default device is used, and so on.

Choosing Default and Non-Default Devices

sp_diskdefault lets you plan space usage carefully for performance and recovery, while allowing users to create or alter databases occasionally.

Make sure these devices are **not** default devices:

- The master device (use **sp_diskdefault** to set **defaultoff** after adding user devices)
- The device for *sybsecurity*
- Any device intended solely for logs
- Devices where high-performance databases reside, perhaps using segments

You can use the device that holds *sybsystemprocs* for other user databases.

➤ ***Note***

If you're using disk mirroring or segments, you should exercise caution in deciding which devices you add to the default list with **sp_diskdefault**.

In most cases, devices that are to be mirrored or databases that will contain objects placed on segments should allocate devices specifically, rather than being made part of default storage.

Disk Mirroring

Disk mirroring can provide non-stop recovery in the event of media failure. The **disk mirror** command causes a SQL Server database device to be "duplicated": all writes to the device are copied to a separate physical device. If one of the devices fails, the other contains an up-to-date copy of all transactions.

➤ ***Note***

You cannot mirror a dump device.

When a read or write to a mirrored device fails, SQL Server causes the bad device to become unmirrored, and displays error messages. SQL Server continues to run, unmirrored. The System Administrator must issue a **disk remirror** command to restart mirroring.

Deciding What to Mirror

When deciding to mirror a device, you must weigh such factors as the costs of system downtime, possible reduction in performance, and the cost of storage media. Reviewing these issues will help you decide what to mirror: just the transaction logs, all devices on a server, or selected devices.

Figure 3-2: Disk Mirroring Using Minimal Physical Disk Space illustrates the "minimum guaranteed configuration" for database recovery in case of hardware failure. The master device and a mirror of the user database transaction log are stored in separate partitions on one physical disk. The other disk stores the user database and its transaction log in two separate partitions.

If the disk with the user database fails, you can restore the user database on a new disk from your backups and the mirrored transaction log.

If the disk with the master device fails, you can restore the master device from a database dump of the *master* database and remirror the user database's transaction log.

This configuration minimizes the amount of disk storage required. It provides for full recovery, even if the disk storing the user database and transaction log is damaged, because the mirror of the transaction log ensures full recovery. However, this configuration does not provide non-stop recovery because the *master* and user databases are not being mirrored, and must be recovered from backups.

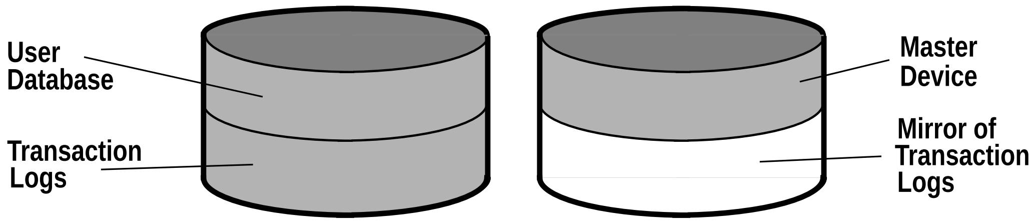


Figure 3-2: Disk Mirroring Using Minimal Physical Disk Space

Figure 3-3: Disk Mirroring for Rapid Recovery represents another mirror configuration. In this case, the master device, user databases, and the transaction log are all stored on different partitions of the same physical device, and are all mirrored to a second physical device.

The configuration in Figure 3-3 provides non-stop recovery from hardware failure. Working copies of the *master* and user databases and log on the primary disk are all being mirrored, and failure of either disk will not interrupt SQL Server users.

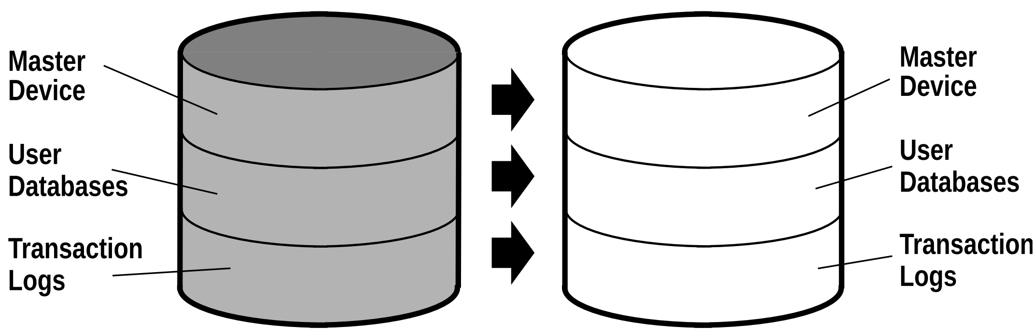


Figure 3-3: Disk Mirroring for Rapid Recovery

With this configuration, all data is written twice, once to the primary disk, and once to the mirror. Applications which involve many writes may be slower with disk mirroring than without mirroring.

Figure 3-4: Disk Mirroring: Keeping Transaction Logs on a Separate Disk illustrates another configuration with a high level of redundancy. In this configuration, all three database devices are mirrored, but the configuration uses four disks instead of two. This configuration speeds performance during write transactions because the database transaction log is stored on a different device from the user databases, and the system can access both with less disk head travel.

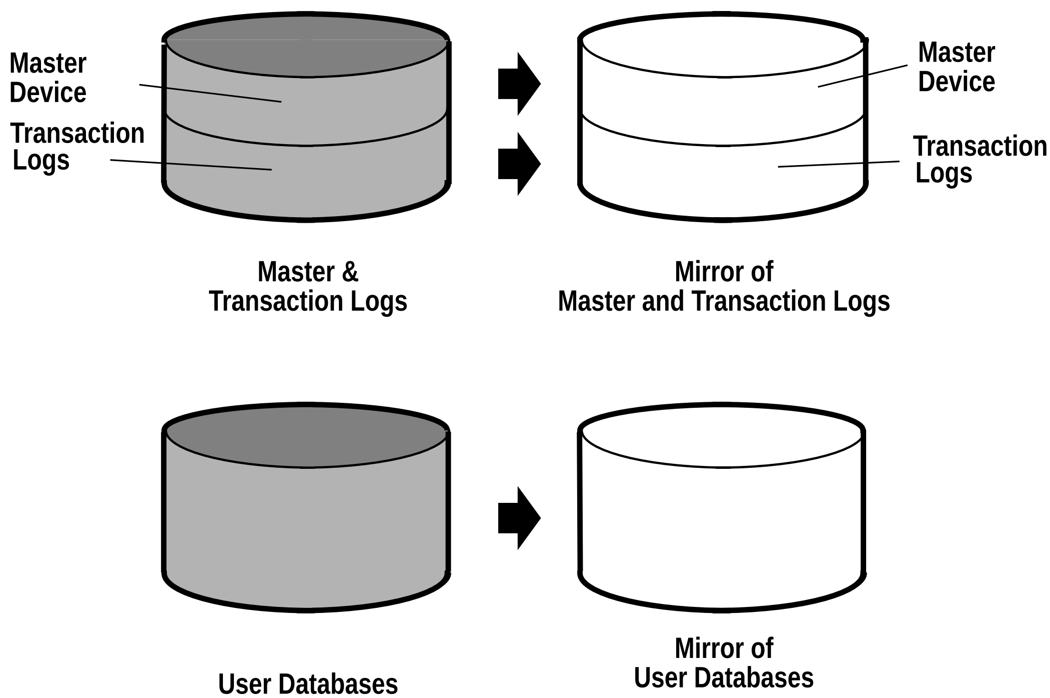


Figure 3-4: Disk Mirroring: Keeping Transaction Logs on a Separate Disk

In summary, these three examples involved different cost and performance trade-offs:

1. **Speed of Recovery.** You can achieve non-stop recovery when the *master* and user databases (including logs) are mirrored, and can recover without the need to reload transaction logs.
2. **Storage space.** Immediate recovery requires full redundancy (all databases and logs mirrored), which consumes disk space.
3. **Impact on performance.** Mirroring the user databases (as in Figure 3-3 and Figure 3-4) increases the time needed to write transactions to both disks.

Disk Mirroring Commands

Three commands, **disk mirror**, **disk unmirror** and **disk remirror**, control disk mirroring. All of the commands can be issued while the devices are in use, so you can start or stop database device mirroring while databases are being used.

> ➤ ***Note***
>
> All of these commands alter the *sysdevices* table in the *master* database. After issuing any of these commands you should dump the *master* database to ensure recovery in case *master* is damaged.

Initializing Mirrors

The **disk mirror** command starts disk mirroring. **Do not** initialize the mirror device with **disk init**. A database device and its mirror constitute one logical device. The mirror name is added to the *mirrorname* column in the *sysdevices* table.

> ➤ ***Note***
>
> To retain use of asynchronous I/O, mirror raw devices to raw devices, and operating system files to operating system files. Mirroring a raw device with a regular file produces an error message. Mirroring a regular file with a raw device will work, but will not use asynchronous I/O.

Here is the **disk mirror** syntax:

```
disk mirror
   name = "device_name" ,
   mirror = "physicalname"
   [ , writes = { serial | noserial }]
   [ , contiguous ]       (OpenVMS only)
```

The *device_name* is the name of the device that you want to mirror, as it is recorded in *sysdevices.name* (by **disk init**). The *physicalname* is a full path name for the mirror disk, which is not yet initialized. It cannot be an already-existing operating system file.

On systems that support asynchronous I/O, the **writes** option allows you to specify whether writes to the first device must finish before writes to the second device begin (**serial**), or whether both I/O requests are to be queued immediately, one to each side of the mirror (**noserial**).

In either case, if a write cannot be completed, the I/O error causes the bad device to become unmirrored.

serial writes are the default. The writes to the devices take place consecutively; the first one finishes before the second one starts. **serial** writes provide protection in the case of power failures: one write may be garbled, but both of them will not be. **serial** writes are generally slower than **noserial** writes.

OpenVMS users should see the *SQL Server Utility Programs* manual for an explanation of the **contiguous** option.

In the following example, *tranlog* is the logical device name for a raw device. The *tranlog* device was initialized with **disk init** and is being used as a transaction log device (as in **create database**... **log on** *tranlog*). The following command mirrors the transaction log device:

```
disk mirror
  name = tranlog,
  mirror = "/dev/rxy1e"
```

Effects on System Tables

The database device that you want to mirror will already have been initialized with **disk init**. The *physicalname* that you give to **disk mirror** is added to the existing row in *sysdevices* in the *mirrorname* column and the *status* bits are updated to reflect the configuration you've chosen.

Unmirroring a Device

Disk mirroring is automatically de-activated when one of the two physical devices fails.

Use the **disk unmirror** command to stop the mirroring process when hardware maintenance is needed or when a hardware device needs to be changed.

```
disk unmirror
  name = "device_name"
  [, side = { "primary" | secondary }]
  [, mode = { retain | remove }]
```

The **side** option to the **disk unmirror** command allows you to specify which side of the mirror to disable. **primary** (in quotes) is the device listed in the *name* column of *sysdevices*; **secondary** (no quotes required) is the device listed in the *mirrorname* column of *sysdevices*. **secondary** is the default.

The **mode** option indicates whether the unmirroring process should be temporary (**retain**) or permanent (**remove**). **retain** is the default.

Effects on System Tables

The **mode** option changes the *status* column in *sysdevices* to indicate that mirroring has been disabled. Its effects on the *phyname* and *mirrorname* columns in *sysdevices* depend on the **side** argument also, as shown in Table 3-3.

<table>
<tr><th></th><th></th><th colspan="2">side</th></tr>
<tr><th></th><th></th><th>primary</th><th>secondary</th></tr>
<tr><td rowspan="2">mode</td><td>remove</td><td>Name in mirrorname moved to phyname and mirrorname set to null; status changed</td><td>Name in mirrorname removed; status changed</td></tr>
<tr><td>retain</td><td colspan="2">Names unchanged; status changed to indicate which device is being de-activated</td></tr>
</table>

Table 3-3: Effects of mode *and* side *Options to* disk mirror *Command*

This example suspends the operation of the primary device:

```
disk unmirror
  name = tranlog,
  side = primary
```

Restarting Mirrors

Use **disk remirror** to restart a mirror process that has been suspended due to a device failure or with **disk unmirror**. The syntax is:

```
disk remirror
     name = "device_name"
```

This command copies the database device to its mirror.

waitfor mirrorexit

Since disk failure can impair system security, the **waitfor mirrorexit** command can be included in an application to perform specific tasks when a disk becomes unmirrored.

```
begin
   waitfor mirrorexit
    commands to be executed
end
```

The commands depend on your applications. You may wish to add certain warnings in applications which perform updates, or use

sp_dboption to make certain databases read only if the disk becomes unmirrored.

➤ ***Note***

SQL Server only knows that a device has become unmirrored when it attempts I/O to the mirror device. On mirrored databases, this occurs at a checkpoint, or whenever the SQL Server buffer must be written to disk. On mirrored logs, I/O occurs whenever a process writes to the log, including any committed transaction that performs data modification, a checkpoint, or a database dump.

waitfor mirrorexit (and the error messages that are printed to the console and errorlog on mirror failure) are only activated by these events.

Mirroring the Master Device

If you choose to mirror the device that contains the *master* database, you need to edit the runserver file for your SQL Server, so that the mirror device starts when the server boots.

On UNIX, add the **-r** flag and the name of the mirror device:

```
dataserver -d /dev/rsd1f -r /dev/rs0e -e/sybase/install/errorlog
```

On OpenVMS, add the mirror name:

```
dataserver /device=(DUA0:[dbdevices]master.dat, -
    DUB1:[dbmirrors]mirror.dat) -
    /errorfile=sybase_system:[sybase.install]errorlog
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Creating User Databases

User databases are created with the **create database** command. All databases must be created while using the *master* database. To create a database, a user must be a valid user of *master* (added with **sp_adduser**) and must have **create database** permission.

The **create database** command clears every page of the database device. It may take up to several seconds or several minutes to complete, 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, you can use an option to **create database** that skips page clearing (the page clearing step will be performed after the load completes).

create database Syntax

The **create database** syntax is:

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

A database name must follow the rules for identifiers. You can create only one database at a time.

In its simplest form, **create database** creates a database on the default devices available on your server:

```
create database newpubs
```

The optional **on** clause allows you to specify the names of one or more database devices, and the space allocation for each database device in megabytes. All of the database devices named in **create database** must be listed in *sysdevices*, that is, they must have been initialized with **disk init**. If you use the keyword **default** (or if you omit the **on** clause altogether), the database is stored on one or more of the default database devices.

To specify a size (in this example, 4 megabytes) for a database to be stored in a default location, use **on default =** ***size*** like this:

```
create database newpubs
on default = 4
```

To place the database on specific database devices, give the name(s) of the database device(s) on which you want it stored. As the syntax indicates, you can request that a database be stored on more than one database device, with different amounts of space on each. The following statement creates the *newdb* database and allocates 3 megabytes on *mydata* and 2 megabytes on *newdata2*. It places the transaction log on a third database device, *tranlog*:

```
create database newdb
on mydata = 3, newdata = 2
log on tranlog = 2
```

How User Databases are Created

When a **create database** statement is issued:

- Verifies that the database name specified in the statement is unique
- Makes sure that the database device names specified in the statement are available
- Finds an unused identification number for the new database
- Assigns space to the database on the specified database devices, and updates *master..sysusages* to reflect these assignments
- Inserts a row into *sysdatabases*
- Makes a copy of the *model* database in the new database space, thereby creating the new database's system tables
- Clears the remaining pages in the database (**create database...for load** skips this step)

The new database initially contains a set of system tables, with entries that describe the system tables themselves. The new database inherits all of the changes you have made to the *model* database. These can include:

- The addition of user names
- The addition of objects
- The database option settings. Originally, the options are set off in *model*. If you want all of your new user databases to inherit particular options, you can change the options in *model* with the system procedure **sp_dboption**.

Permissions for Creating Databases

Only System Administrators can grant permission to use the **create database** command. In many installations, System Administrators maintain a monopoly on **create database** permission in order to centralize control of database placement and database device allocation.

In these situations, System Administrators create new databases on behalf of other users, and then transfer ownership to the appropriate user.

To create a database that is to be owned by another user, the System Administrator:

1. Issues the **create database** command,
2. Switches to the new database with the **use** command, and then
3. Executes the system procedure **sp_changedbowner**.

The System Administrator can also directly **grant** permission to create databases.

The fact that System Administrators seem to operate outside the protection system serves as a safety precaution. For example, if a Database Owner forgets his or her password or accidentally deletes all entries in *sysusers*, a System Administrator can repair the damage (from the backups or dumps that are, of course, regularly made).

Assigning Databases to Database Devices

Databases are allocated storage space when the **create database** or **alter database** command is issued. The **create database** command can specify one or more database devices, and the amount of space on each that is to be allocated to the new database. In addition, the **log on** clause to **create database** should be used to place the database's transaction log on a separate device.

If no database device is specified (or if the keyword **default** is used), SQL Server puts the database on one or more of the default database devices specified in *master..sysdevices*.

Database Size

If the size parameter in the **on** clause is omitted, the database is created with the default amount of space. This amount will be the greater of:

- The **database size** parameter in *master..sysconfigures*, or
- The size of the *model* database.

The size of *model* and the value of **database size** are initially set to 2 megabytes. The size of *model* can be changed by allocating more space with **alter database**. The **database size** configuration variable can be reset with **sp_configure**. The default size for new databases can be changed to any size greater than 2 megabytes (up to 10,000 megabytes), by updating *master..sysconfigures* with the stored procedure **sp_configure** and issuing the **reconfigure** command. See Chapter 12, ‘‘Fine-Tuning Performance and Operations’’, for complete instructions.

If the amount of space you request on a specific database device is unavailable, SQL Server creates the database with as much space as possible on a per-device basis. It then displays a message informing you how much space was actually allocated on each database device. (This is not considered an error.) If there is less than the minimum space necessary for a database on the specified database device (or on the default, if you don't give any names), the **create database** command fails.

The allocation decisions that you make when you issue **create database** or **alter database** are important, because it is difficult to reclaim storage space once it's assigned. You can always add additional space; however you cannot deallocate space that has been assigned to a database unless you drop the database first.

Estimating the Size of Tables and Indexes

You can estimate the size of the tables and indexes for your database using the **sp_estspace** system procedure. Before you create your user database, follow these steps for each table:

1. Create the table, with all of the columns and datatypes you will use. Both the length specification and the nulltype of the columns is important in determining the amount of storage space a table will occupy.
2. Create all of the indexes on the table.
3. Execute **sp_estspace**, giving the table name and the number of rows that you estimate the table will hold.

You can perform this work in a test or development database, or in *tempdb*.

sp_estspace prints the estimated size of the table and each of its indexes, and the sum of these sizes.

Add the results from **sp_estspace** to get an estimate of the size of the user tables and indexes. Each database also requires additional space for:

- *syslogs*. If you store your database's log on a separate segment using the **log on** clause, you don't have to include that in your size estimate for the data portion of your database.
- Views, procedures, defaults, rules, and triggers you create. System tables store a copy of the text of these objects, and a copy of the compiled procedure plan.

- Other system tables. Execute **sp_helpsegment system** in the *model* database or a newly created database to see how much free space is available on the segment that stores user and system tables. Adding users, granting permissions, and performing other actions in the database that affect the system tables increase the amount of space these tables use.

See **sp_estspace** in Volume 2 of the *SQL Server Reference Manual* for more information on using this procedure.

Omitting the *on* Clause

If you omit the **on** clause completely, the size of the database is the default size, as described. The space is allocated from the default database devices indicated in *master..sysdevices*, in alphabetical order by database device name.

Use the following query to see the logical names of default database devices:

```
select name
    from sysdevices
    where status & 1 = 1
    order by name
```

sp_helpdevice also displays "default disk" as part of the description of database devices.

Placing the Transaction Log on a Separate Device: *log on*

The **log on** clause to **create database** places the transaction log (the *syslogs* table) on a separate database device. If the size parameter in the **log on** clause is omitted, the log device is allocated 2 megabytes of storage.

There are several reasons to place the logs on a separate database device:

- It allows you to use the **dump transaction** command (rather than **dump database**), thus saving time and tapes.
- It allows you to establish a fixed size for the log, keeping it from competing with other database activity for space.
- It creates default free-space threshold monitoring on the log segment, and allows you to create additional free-space monitoring on the log and data portions of the database.

- It improves performance.
- It ensures full recovery in the event of hard disk crashes. A special argument to **dump transaction** lets you dump your transaction log, even if your data device is on a damaged disk.

Unless you are creating very small, non-critical databases, you should always use the **log on** clause to **create database**.

Determining the Size of the Transaction Log

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, whether you perform the dumps manually or use threshold procedures to dump your transaction logs automatically. As a rule of thumb, allocate to the log 10% to 25% of the space you allocate to the database itself.

Inserts, deletes and updates increase the size of the logs, and **dump transaction** decreases the size of the log by writing committed transactions to disk and removing them from the log. Since **update** statements require logging of both the "before" and "after" images of a row, applications which require mass updates of large numbers of rows should plan for transaction log space at least twice as large as the number of rows to be updated at once—or twice as large as your largest table. (Of course, you can also "batch" the updates in smaller groups with transaction dumps between the batches.)

In databases with a lot of insert and update activity, logs can grow quickly. Periodic checking can help you determine the correct size for your log, can help you choose thresholds for the log, and can help you schedule transaction log dumps. To check on the space used by your transaction log, you must be using the database. Then, issue the command:

```
dbcc checktable(syslogs)
```

dbcc reports the number of data pages in use by the log. If your log is on a separate device, **dbcc checktable** also tells you how much space is used and free. Here is sample output for a 2 megabyte log:

```
Checking syslogs
The total number of data pages in this table is 199.
*** NOTICE: Space used on the log segment is 0.39 Mbytes, 19.43%.
*** NOTICE: Space free on the log segment is 1.61 Mbytes, 80.57%.
Table has 1661 data rows.
```

Another command you can use to check on the growth of the log is:

```
select count(*) from syslogs
```

You can repeat either command periodically to see how fast the log grows.

Omitting the log on Clause

If you omit the **log on** clause, the database's transaction log is placed on the same database device as the data tables. Subsequent use of the system procedure **sp_logdevice** (as described on page 3-28) affects only future writes to the log, and does not immediately move the first few log pages that were written when the database was created. This leaves exposure problems in certain recovery situations, and is not recommended.

➤ ***Note***

Each time you issue the **create database** command, you should dump the *master* database. This makes recovery easier and safer in case *master* is later damaged. See Chapter 9, "Backing Up and Restoring the System Databases".

for load Option for Database Recovery

Use the **for load** option if you are going to use the database for loading from a database dump, either for recovery from media failure or for moving a database from one machine to another. Using **for load** runs a streamlined version of **create database** to create a target database that can **only** be used for loading a dump.

If 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**

When you load a database dump, the new database device allocations for the database need to match the usage allocations of the dumped database. See Chapter 8, "Backing Up and Restoring User Databases", for a discussion of duplicating space allocation.

After you load the database dump into the new database, there are no restrictions on the commands you can use.

with override Option to *create database*

This option allows SQL Servers on machines with limited space to still maintain their logs on separate device fragments from their data. Use this option **only** when you're putting log and data on the same logical device. This is not recommended practice, but may be the only option available on machines with limited storage, especially if you need to get databases back on-line following a hard disk crash.

You'll still be able to dump your transaction log, but if you experience a media failure, you won't be able to access the current log (since it's on the same device as the data). You will be able to recover only to the last transaction log dump, and all transactions between that point and the failure time will be lost.

Here is an example where log and data are on separate fragments of the same logical device:

```
create database littledb
    on diskdev1 = 4
    log on diskdev1 = 1
    with override
```

Use this option only if you are not concerned with up-to-the-minute recoverability.

Moving the Transaction Log to Another Device

If you did not use the **log on** clause to **create database**, the following procedure allows you to move your transaction log to another database device.

The system procedure **sp_logdevice** moves future allocation for a database's transaction log. However, the transaction log remains on the original device until the allocated page has been filled, and the transaction log has been dumped.

Here is the syntax for **sp_logdevice**:

sp_logdevice *database_name*, *device_name*

The database device that you name must be initialized with **disk init**, and must be allocated to the database with **create** or **alter database**.

To move the entire transaction log, complete these steps:

1. Execute **sp_logdevice**, naming the new database device.
2. Execute enough transactions to fill the page that is currently in use. Since the page contains 2048 bytes, you may need to update

at least 2048 bytes. You can execute **dbcc checktable(syslogs)** before and after you start updating to determine when a new page is used.

3. Wait for all currently active transactions to finish to ensure that there are no active transactions on the database device. You may want to perform this entire activity after putting the database into single-user mode with **sp_dboption**.
4. Run **dump transaction**. See Chapter 7, ‘‘Developing a Backup and Recovery Plan’’, for more information. **dump transaction** removes all the log pages that it writes to disk; as long as there are no active transactions in the part of the log on the old device, all of those pages will be removed.
5. Run the system procedure **sp_helplog** to ensure that the complete log is on the new log device.

➤ ***Note***

When you move a transaction log, the space no longer used by the transaction log becomes available for data. You cannot reduce the amount of space allocated to a device by moving the transaction log.

Transaction logs are discussed in detail in Chapter 7, ‘‘Developing a Backup and Recovery Plan’’.

Changing Database Ownership

The system procedure **sp_changedbowner** can be used to change the ownership of a database. A System Administrator might want to create the user databases, giving ownership of them to another user after some of the initial work has been completed. The procedure must be executed by the System Administrator in the database whose ownership will be changed. The syntax is:

sp_changedbowner *loginame* [, true]

This example makes the user “albert,” the owner of the current database and drops aliases of users who could act as the old “dbo”:

```
sp_changedbowner albert
```

The new owner must already have a login name on SQL Server, but cannot be a user of the database, or have an alias in the database. You may have to use **sp_dropuser** or **sp_dropalias** before you can change a database’s ownership.

To transfer aliases and their permissions to the new Database Owner, add the second parameter with the value "true" or "TRUE".

➤ ***Note***

You cannot change the ownership of the *master* database. It is always owned by the "sa" login.

Increasing the Size of a Database

When your database grows to fill all of the space you allocated with **create database**, you use the **alter database** command to add additional storage. You can add space for database objects or transaction log space, or both at once. **alter database** is also used to prepare to load a database from backup.

Permission to use the **alter database** command defaults to the Database Owner, and permission is automatically transferred with database ownership (see **sp_changedbowner** in the *SQL Server Reference Manual*). **alter database** permission cannot be changed with **grant** or **revoke**.

If you execute **alter database** with only a database name, it adds one megabyte on the default database devices. If your database separates log and data, the space you add will be entirely used for data. You can find names of database devices that are in your default list with **sp_helpdevice**.

To add one megabyte on a default database device for the *newpubs* database, type:

```
alter database newpubs
```

alter database Syntax

The full syntax allows you to extend a database by other amounts and to specify where storage space is to be added:

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

The **on** and **log on** clauses in the **alter database** command are just like the corresponding clauses in the **create database** command. You can specify space on a default database device or on some other database device, and you can name more than one database device. If you are using **alter database** to extend the *master* database, however, you can only extend it on the master device. The minimum increase you can specify is one megabyte (512 2K pages).

To add three megabytes to the space allocated for the *newpubs* database on the database device named *pubsdata1*, the command is:

```
alter database newpubs
on pubsdata1 = 3
```

If SQL Server can't allocate the requested size, it allocates as much as it can on each database device with a minimum allocation of one-half megabyte (256 2K pages) per device. When **alter database** completes, it prints messages telling you how much space it allocated, like this:

```
Extending database by 1536 pages on disk pubsdata1
```

You should check these messages to be sure you added enough space.

Here is how to add two megabytes to the space allocated for *newpubs* on *pubsdata1*, and add three megabytes on a new device, *pubsdata2*, plus another megabyte for the log, on *tranlog*:

```
alter database newpubs
on pubsdata1 = 2, pubsdata2 = 3
log on tranlog
```

➤ ***Note***

Each time you issue the **alter database** command, dump the *master* database.

The with override *Clause*

The **with override** clause allows you to create a device fragment containing log space on a device which already contains data, or a data fragment on a device already in use for the log. Use this option only in cases where you have no other storage options, and where up-to-the-minute recoverability is not critical.

The for load *Clause*

The **for load** option is used only after **create database for load** to recreate the space allocation of the database being loaded into the new database from a dump. See Chapter 8, "Backing Up and Restoring

User Databases'', for a discussion of duplicating space allocation when loading a dump into a new database.

drop database Command

The **drop database** command removes a database from the server, deleting the database and all of the objects in it from SQL Server. This command:

- Frees the storage space allocated for the database
- Deletes references to it from the system tables in the *master* database

The syntax of this command is:

drop database *database_name* [, *database_name*]...

Only the Database Owner can drop a database. You must be in the *master* database to drop any database. You cannot drop a database that is in use (open for reading or writing by any user).

You can drop more than one database in a single statement. For example:

```
drop database newpubs, newdb
```

If tables in a database are referenced by referential integrity constraints from other databases, you must remove the constraints before you can drop the database. Use **alter table** to drop the constraints.

You must **drop** all the databases on a database device before you can drop the database device. The command to drop a device is **sp_dropdevice**.

After you drop a database, dump the *master* database to ensure recovery in case *master* is damaged.

How SQL Server Allocates Space for a Database

For the user, creating a database on a database device and allocating a certain amount of space to it is just a matter of issuing a command. For SQL Server, it's a more involved task.

SQL Server first makes an entry for the new database in *sysdatabases*. Then it checks *master..sysdevices* to make sure that the device names specified in the **create database** command actually exist and are database devices. If you did not specify database devices, or used the

default option, SQL Server checks *master..sysdevices* and *master..sysusages* for free space on all devices that can be used for default storage. It performs this check in alphabetical order by device name.

The storage space from which SQL Server gathers the specified amount of storage need not be contiguous, and can be extracted from whatever free space is available. The database storage space can even be drawn from more than one database device. Of course, a database is treated as a logical whole even if it is stored on more than one database device.

Each piece of storage for a database must be at least 1 allocation unit—half a megabyte, or 256 contiguous 2K pages (on Stratus, 256 4K pages).The first page of each allocation unit is the allocation page. It does not contain database rows like all the other pages. Rather, it contains an array that shows how the other 255 pages are used.

The *sysusages* Table

The database storage information is listed in the table *master..sysusages*. Each row in *master..sysusages* represents a space allocation assigned to a database. Thus, each database has one row in *sysusages* for each time **create database** or **alter database** assigns a fragment of disk space to a database.

When SQL Server is first installed, *sysusages* contains rows for these *dbids*:

- 1, the *master* database
- 2, the temporary database, *tempdb*
- 3, the *model* database
- 4, the *sybsystemprocs* database

If you installed auditing, the *sybsecurity* database will be *dbid* 5.

➤ ***Note***

If your installation is an upgrade from a pre-Release 10.0 SQL Server, *sybsystemprocs* and *sybsecurity* may have different database IDs.

As new databases are created or current databases enlarged, new rows are added to *sysusages* to represent new database allocations.

Here is what *sysusages* might look like on a SQL Server with the five system databases and two user databases (with *dbid*s 6 and 7). Both of these databases were created with the **log on** option. The database with *dbid* 7 has been given additional storage space with two **alter database** commands:

```
select dbid, segmap, lstart, size, vstart
from sysusages
```

```
dbid   segmap      lstart      size            vstart
------ ----------- ----------- -----------    -------
     1           7           0        1536          4
     2           7           0        1024       2564
     3           7           0        1024       1540
     4           7           0        5120   16777216
     5           7           0       10240   33554432
     6           3           0         512    1777216
     6           4         512         512    3554432
     7           3           0        2048   67108864
     7           4        2048        1024   50331648
     7           3        3072         512   67110912
     7           3        3584        1024   67111424

(10 rows affected)
```

The segmap *Column*

The *segmap* column is a bitmap linked to the *segment* column in the user database's *syssegments* table. Since the *logsegment* in each user database is segment 2, and these user databases have their logs on separate devices, *segmap* contains 4 (2^2) for those devices named in the **log on** statement, and 3 for the data segment that holds the system segment ($2^0 = 1$) + default segment ($2^1 = 2$).

Here are some of the possible values for segments containing data or logs:

Value	Segment
3	data only (system and default segments)
4	log only
7	data and log

Table 3-4: Segment Values

Values higher than 7 indicate user-defined segments. The *segmap* column is explained more fully in the segments tutorial section, later in this chapter.

The lstart, vstart *and* size *Columns*

The *lstart* column contains the starting page number within the database of this allocation unit. Each database starts at logical address "0." If additional allocations have been made for a database, as in the case of *dbid* 7, the *lstart* field reflects this.

The *size* column contains the number of contiguous 2K pages that are assigned to the same database (on Stratus, 4K pages). The ending logical address of this portion of the database can be determined by adding the values in *lstart* and *size*.

The *vstart* column contains the address where the piece assigned to this database begins. The upper four bits store the virtual device number (*vdevno*), and the lower four bits store the virtual block number. (To obtain the virtual device number, divide *sysusages.vstart* or *sysdevices.low* by 16,777,216, which is 2^{24}.) The value in *vstart* identifies which database device contains this portion of the database, because it falls between the values in the *low* and *high* columns of *sysdevices* for the database device in question.

Creating and Using Segments

Segments are named subsets of the database devices available to a particular SQL Server database. A segment can best be described as a label that points to one or more database devices. Segment names are used in **create table** and **create index** commands to place tables or indexes on specific database devices. The use of segments can increase SQL Server performance, and can give the System Administrator or Database Owner increased control over placement, size and space usage of specific database objects. For example:

- If a table is placed on one device, and its non-clustered indexes on a device on another disk controller, the time required to read or write to the disk can be reduced, since disk head travel is usually reduced.
- If a large, heavily-used table is split across devices on two separate disk controllers, read / write time may be improved.
- SQL Server stores the data for *text* and *image* columns on a separate chain of data pages. By default, this text chain is placed on the same segment as the table. Since reading a text column requires a read operation for the text pointer in the base table and an additional read operation on the text page in the separate text chain, placing the text chain on a separate physical device can improve performance.

- If you place tables and indexes only on specific segments, those database objects cannot grow beyond the space available to the devices represented by those segments, and other objects cannot contend for space with them. Segments can be extended to include additional devices as needed.
- You can use thresholds to warn you when space becomes low on a particular database segment. See Chapter 10, “Managing Free Space with Thresholds” for more information.

Segments are created within a particular database from the database devices already allocated to that database. Each SQL Server database can contain up to 32 segments. The database devices must first be initialized with **disk init**, and then be made available to the database with a **create database** or **alter database** statement before you can assign segment names.

When you first create a database, SQL Server creates three segments in the database:

Segment	Function
system	Stores this database's system tables.
logsegment	Stores this database's transaction log.
default	Stores all other database objects—unless you create additional segments, and store the table or index on the new segments by using **create table... on *segment_name*** or **create index... on *segment_name***.

Table 3-5: Default Segments

Commands and Procedures for Using Segments

SQL Server procedures and commands for using segments are:

Command	Action
sp_addsegment	Define a segment in a database
create table and **create index**	Create database objects on segments
sp_dropsegment	Remove a segment from a database
sp_extendsegment	Add additional devices to an existing segment

Table 3-6: Commands and Procedures for Managing Segments

Command	Action
sp_placeobject	Assign future space allocations for a table or index to a specific segment
sp_helpsegment	Display segment allocation for a database or data on a particular segment
sp_helpdb	Display the segments on each database device

Table 3-6: Commands and Procedures for Managing Segments

The following sections introduce the system procedures and commands for using segments, and review how segments affect system tables. The final section on segments is a tutorial showing how to use the system procedures and commands. See Chapter 12, "Fine-Tuning Performance and Operations", for another example of using segments to improve system performance.

Creating Segments

There are two preliminary steps to creating a segment within a database:

- Initialize the physical device with **disk init**.
- Make the database device available to the database by using the **on** clause to **create database** or **alter database**.

Once the database device exists and is available to the database, define the segment in the database with the stored procedure **sp_addsegment**. Here is the syntax:

```
sp_addsegment segname, dbname, devname
```

segname can be any valid identifier. It will be used in **create table** and **create index** statements to place those objects on the segment (and thereby, on the device named in *devname*).

dbname is the name of the database where the next segment will be created.

devname is the name of the database device: the name used in **disk init** and **create** and **alter database** statements.

This statement creates the segment *seg_mydisk1* on the database device *mydisk1*:

```
sp_addsegment seg_mydisk1, mydata, mydisk1
```

Creating Database Objects on Segments

After the segment has been defined in the current database, the **create table** or **create index** commands use the optional clause **on *segment_name*** to place the object on a segment. Here is the syntax:

```
create table table_name (col_name datatype ... )
   [on segment_name]
```

```
create [ clustered | nonclustered ] index index_name
   on table_name(col_name)
   [on segment_name]
```

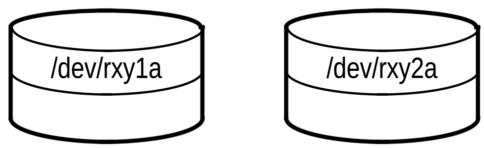
➤ ***Note***

Clustered indexes, where the bottom or leaf level of the index contains the actual data, are by definition on the same segment as the table. See “Segments and Clustered Indexes” on page 3-48.

Figure 3-5: Placing Objects on Specific Devices Using Segments shows the sequence of Transact-SQL commands used to place specific database objects on specific physical disks:

1. Initialize the physical disks.
2. Allocate them to a database.
3. Label the devices with segment names.
4. Create the objects, giving the segment name.

Physical devices



Select physical devices to be used by SQL Server.

use *master*

*Start in **master** database.*

disk init **name = "**mydisk1**",** **physname = "**/dev/rxy1a**",** **vdevno =** 7**,** **size =** 2048	**disk init** **name** = "mydisk2", **physname** = "/dev/rxy2a", **vdevno** = 8, **size** = 1024

*Map SQL Server database device name to physical device with **disk init***

alter database mydata
on mydisk1 = 4, mydisk2 = 2

*Add the devices **mydisk1** and **mydisk2** to **mydata**.*

use mydata

*Change to **mydata** database.*

sp_addsegment seg_mydisk1, mydata, mydisk1
sp_addsegment seg_mydisk2, mydata, mydisk2

Map segment names to database device names.

sp_dropsegment "default", mydata, mydisk1
sp_dropsegment system, mydata, mydisk1
sp_dropsegment "default", mydata, mydisk2
sp_dropsegment system, mydata, mydisk2

*Drop **system** and **default** segments from devices.*

create table authors (au_id etc.) **on** seg_mydisk1
create nonclustered index au_index **on** authors (au_id)
on seg_mydisk2

Create table on one segment, and create its index on the other segment.

Figure 3-5: Placing Objects on Specific Devices Using Segments

Extending Segments

The stored procedure **sp_extendsegment** increases the size of a segment, including additional database devices as part of an existing segment. The syntax is:

sp_extendsegment *segname*, *dbname*, *devname*

Before you can extend a segment:

- The database device must be listed in *sysdevices* (through the use of **disk init**),
- The database device must be available in the desired database (through an **alter database** or **create database** statement), and

- The segment name must exist in the current database (through earlier use of **sp_addsegment**).

The following example adds the database device *pubs_dev2* to an existing segment named *bigseg*:

```
sp_extendsegment bigseg, pubs2, pubs_dev2
```

Since the word "default" is a keyword, if you want to extend the default segment in your database, you must place the word default in quotes:

```
sp_extendsegment "default", mydata, newdevice
```

Placing Objects on a Segment

The system procedure **sp_placeobject** does not remove an object from its allocated segment; however, it causes all further disk allocation for that object to occur on the segment it specifies. Here is the syntax:

sp_placeobject ***segment_name*****,** ***object_name***

The following command causes all further disk allocation for the *mytab* table to take place on *bigseg*:

```
sp_placeobject bigseg, mytab
```

➤ ***Note***

sp_placeobject does **not** move an object from one database device to another. Whatever pages have been allocated on the first device remain allocated; whatever data was written to the first device remains on the device. **sp_placeobject** affects only future space allocations.

To completely move a table, you can drop its clustered index (if one exists), and create or re-create a clustered index on the desired segment. To completely move a non-clustered index, drop the index and re-create it on the new segment.

After you have used **sp_placeobject**, executing **dbcc checkalloc** causes the following message to appear for each object that is split across segments:

```
Extent not within segment: Object object_name,
indid index_id includes extents on allocation page
page_number which is not in segment segment_name.
```

You can ignore these messages.

Dropping Segments

The stored procedure **sp_dropsegment**, when used with only a segment name and the database name, drops the named segment from a database. You cannot drop a segment as long as it contains database objects. You must assign the objects to another segment (as described in the previous note) or you must drop the objects, then drop the segment. The syntax is:

sp_dropsegment *segment_name, dbname*

You can also use **sp_dropsegment** to remove a database device from a segment, reversing the effects of **sp_extendsegment**. This syntax is:

sp_dropsegment *segment_name, dbname, devname*

With three arguments, **sp_dropsegment** does not drop the segment, it just drops the given *devname* from the range of devices that the segment spans. The following example removes the database device *pubs_dev2* from *bigseg*:

```
sp_dropsegment bigseg, pubs2, pubs_dev2
```

➤ ***Note***

Dropping a segment removes its name from the list of segments in the database, but it does not remove the database device from the allocation for that database, nor does it remove objects from the device.

If you drop all of the segments from a database device, the space is still allocated to the database, but cannot be used for database objects. **dbcc checkcatalog** reports "Missing segment in Sysusages segmap." Use **sp_addsegment "default"**, ***dbname, devname*** to map the device to the default segment if you don't plan to assign it to another segment.

Getting Information about Segments

Two system procedures provide information about segments. **sp_helpsegment** lists the segments in the database in which it is executed, or displays information about a particular segment in the database. **sp_helpdb** displays information about the relationship between devices and segments in a database.

sp_helpsegment

The system procedure **sp_helpsegment**, when used without an argument, displays information about all of the segments in the database where you execute it:

```
sp_helpsegment
```

```
segment name                          status
------- ---------------------------- ------
      0 system                            0
      1 default                           1
      2 logsegment                        0
      3 seg1                              0
      4 seg2                              0
```

For information about a particular segment, specify the segment name as an argument. Use quotes when requesting information about the *default* segment:

```
sp_helpsegment "default"
```

The following example displays information about *seg1*:

```
 segment name                    status
------- --------------------- ------
      3 seg1                        0

device                 size          free_space
---------------------- ------------- -----------
seg_mydisk1            2.0MB                1008

table_name            index_name           indid
---------------------- ------------------- ------
authors                au_ind                   1
```

In addition to the segment's number and name, **sp_helpsegment** displays its *status* (1 for the *default* segment, 0 for others). The names of the database devices and their sizes are displayed on the next line. The final line displays information about the tables and indexes on a segment. In this case, the segment stores the index *au_ind* for the *authors* table. The *indid* indicates that it is a clustered index.

sp_helpdb

When you execute **sp_helpdb** within a database, and give that database's name, the system procedure displays information about the segments in the database:

```
sp_helpdb mydata
```

```
name      db_size    owner  dbid created        status
--------- ---------- ----------- ------------- -----------
mydata        8.0 MB sa        4 May 27, 1993   no options set

device_fragments    size       usage            free kbytes
------------------ ---------- ---------------- -----------
datadev2           4.0 MB     data only               3408
logdev             2.0 MB     log only                2032
seg_mydisk1        2.0 MB     data only               2016

device                        segment
----------------------------- -------------------------
datadev2                      default
datadev2                      system
logdev                        logsegment
seg_mydisk1                   seg1
```

Segments and System Tables

Three system tables store information about segments: the *master..sysusages* table, and two system tables in the user database, *sysindexes* and *syssegments*. The system procedure **sp_helpsegment** uses these three tables to provide its information. In addition, it finds the database device name in *sysdevices*.

When you allocate a device to a database with **create database** or **alter database**, SQL Server adds a row to *master..sysusages*. The *segmap* column in *sysusages* provides bitmaps to the segments in the database for each device.

create database also creates the *syssegments* table in the user database with these default entries:

```
segment name             status
 ------- -------------- ------
       0 system              0
       1 default             1
       2 logsegment          0
```

When you add a segment to a database with **sp_addsegment**, the procedure:

- Adds a new row to the *syssegments* table in the user database, and
- Updates the *segmap* in *master..sysusages*.

When you create a table or index, SQL Server adds a new row to *sysindexes*. The *segment* column in that table stores the segment number, showing where the server will allocate new space for the object. If you do not specify a segment name when you create the object, it is placed on the *default* segment; otherwise it is placed on the specified segment. If you create a table containing *text* or *image* columns, a second row is also added to *sysindexes* for the linked list of text pages; by default, the chain of text pages is stored on the same segment as the table. An example using **sp_placeobject** to put the text chain on its own segment is included later in this chapter.

The *name* from *syssegments* is used in **create table** and **create index** statements. The *status* column indicates which segment is the default segment.

A Segment Tutorial

Segments provide a flexible tool for allowing System Administrators to assign objects to particular database devices. For example, you can add a database device to a database, and improve your system performance by assigning a particular high-use table to the device. To achieve these improvements, you must be certain that no other database objects use the new segment. The following tutorial shows how to create a user segment, and how to remove all other segment mappings from the device.

These are important facts to keep in mind when considering how SQL Server handles segments and devices:

- If you assign the space in fragments, for example, some of it with **create database**, and some later pieces with **alter database**, each fragment will have an entry in *sysusages*.
- When an additional fragment of a device is assigned to a database which has already been assigned a fragment of that device, all the segments mapped to the existing fragment are mapped to the new fragment.
- If you use **alter database** to add space on a device that is new to the database, the *system* and *default* segments are mapped to the new space.

The tutorial begins with a new database, created with one device for the database objects, and another for the transaction log:

```
create database mydata on bigdevice = 4
    log on logdev = 2
```

Now, if you **use mydata**, and run **sp_helpdb**, you'll see:

```
sp_helpdb mydata
```

```
name       db_size  owner     dbid   created      status
---------- -------- --------- ------ ------------ ---------------
mydata       6.0 MB sa             4 May 27, 1993  no options set

device_fragments       size          usage           free kbytes
---------------------- ------------- --------------- -----------
bigdevice              4.0 MB        data only              3408
logdev                 2.0 MB        log only               2032

device                 segment
---------------------- ----------------------
bigdevice              default
bigdevice              system
logdev                 logsegment
```

Like all newly created databases, *mydata* has the segments named *default*, *system* and *logsegment*. Since the **create database** statement used **log on**, the *logsegment* is mapped to its own device—*logdev*, and the *default* and *system* segment are both mapped to *bigdevice*.

If you add additional space on the same devices to *mydata*, and then run **sp_helpdb** again, you'll see entries for the added fragments:

```
use master

alter database mydata on bigdevice = 2
    log on logdev = 1

use mydata

sp_helpdb mydata
```

```
name       db_size  owner     dbid   created      status
---------- -------- --------- ------ ------------ ---------------
mydata       9.0 MB sa             4 May 27, 1993  no options set

device_fragments       size          usage           free kbytes
---------------------- ------------- --------------- -----------
bigdevice              2.0 MB        data only              2048
bigdevice              4.0 MB        data only              3408
logdev                 1.0 MB        log only               1024
logdev                 2.0 MB        log only               2032

device                 segment
---------------------- ----------------------
bigdevice              default
bigdevice              system
logdev                 logsegment
```

Always add log space to log space, and data space to data space. SQL Server will warn you, and instruct you to use **with override** if you try to allocate a segment that is already in use for data to the log, or vice versa. Remember that segments are mapped to entire devices, and not just to the space fragments. If you change any of the segment assignments on a device, you make the change for all of the fragments.

When you allocate a new device to the database with **alter database**—one that's not in use by the database—the new fragments are automatically assigned:

- To the *system* and *default* segments, if they are in the **on** clause, or if they are default devices, or
- To the log segment, if they are in the **log on** clause

The following example allocates a new database device that has not been used by *mydata*:

```
use master
alter database mydata on newdevice = 3
use mydata
sp_helpdb mydata
```

```
name       db_size  owner     dbid   created      status
---------- -------- --------- ------ ------------ ---------------
mydata      12.0 MB sa            4 May 27, 1993  no options set

device_fragments       size          usage           free kbytes
---------------------- ------------- --------------- -----------
bigdevice              2.0 MB        data only              2048
bigdevice              4.0 MB        data only              3408
logdev                 1.0 MB        log only               1024
logdev                 2.0 MB        log only               2032
newdevice              3.0 MB        data only              3072

device                 segment
---------------------- ----------------------
bigdevice              default
bigdevice              system
logdev                 logsegment
newdevice              default
newdevice              system
```

The *default* and *system* segments are mapped to the new space. In some cases, you want the new space for use as default storage space for **create table** or **create index** statements. In that case, this allocation is fine. However, if you're adding space in order to assign a table or index to a specific segment (and therefore to specific devices), drop the *system* and *default* segments from the new fragment, and add your own segment name.

The following example creates a segment called *new_space* on *newdevice*:

```
sp_addsegment new_space, mydata, newdevice
```

Here is just the portion of the **sp_helpdb** report that has changed, the portion that lists the segment mapping:

```
device                         segment
------------------------------ ------------------
bigdevice                      default
bigdevice                      system
logdev                         logsegment
newdevice                      default
newdevice                      new_space
newdevice                      system
```

Notice that the *default* and *system* segments are still mapped to *newdevice*. If you're planning to use *new_space* to store a user table or index for improved performance, and want to insure that other user objects are not stored on the device by default, drop the *default* and *system* segments with **sp_dropsegment**:

```
sp_dropsegment system, mydata, newdevice
sp_dropsegment "default", mydata, newdevice
```

The quotes around "default" are needed because it's a Transact-SQL reserved word.

Here is just the portion of the **sp_helpdb** report that shows the segment mapping:

```
device                         segment
------------------------------ --------------------
bigdevice                      default
bigdevice                      system
logdev                         logsegment
newdevice                      new_space
```

Only *new_space* is now mapped to *newdevice*. Users who create objects can use **on** *new_space* to place a table or index on the device that corresponds to that segment. Since the *default* segment is not using that database device, users who create tables and indexes without using the **on** clause won't be placing them on your specially-prepared device.

If you **alter database** on *newdevice* again, the new space fragment acquires the same segment mapping as the existing fragment of that device (that is, the *new_space* segment only).

At this point, if you use **create table** and name *new_space* as the segment, you'll get results like these from **sp_help** and **sp_helpsegment**:

```
create table mytabl (c1 int, c2 datetime)
    on new_space

sp_help mytabl

Name              Owner             Type
----------------- ----------------- ----------------
mytabl            dbo               user table

Data_located_on_segment        When_created
------------------------------ --------------------
new_space                      May 27 1993  3:21PM

Column_name   Type      Length Nulls Default_name Rule_name
------------- --------- ------ ----- ------------ ----------
c1            int            4     0 NULL           NULL
c2            datetime       8     0 NULL           NULL
Object does not have any indexes.
No defined keys for this object.

sp_helpsegment new_space

segment name                           status
------- ------------------------------ ------
      3 new_space                           0

device                  size           free_space
----------------------- -------------- -----------
newdevice               3.0MB                 1528

table_name            index_name             indid
--------------------- ---------------------- ------
mytabl                mytabl                      0
```

Other Factors in Using Segments

Segments and Clustered Indexes

The bottom or leaf level of a clustered index contains the actual data. By definition, therefore, a table and its clustered index are on the same segment. If you create a table on one segment, and then create its clustered index on a different segment, the table travels with its index. The entire table will “leave” the segment on which the table was created and migrate to the segment where the clustered index is created. This provides a quick and easy mechanism for moving a table to another segment in your database.

This example creates a clustered index, without specifying the segment name, using the same table just created on the *new_space* segment in the example above. Checking *new_space* after the **create index** command shows that there are no longer any objects on the segment:

```
/* Don't try this at home */
create clustered index mytabl_cix
    on mytabl(c1)

sp_helpsegment new_space

segment name                          status
------- ----------------------------- ------
      3 myseg                              0

device             size            free_space
------------------ --------------- -----------
datadev2           2.0MB                  1016
```

If you have placed a table on a segment, and need to create a clustered index, be sure to use the **on *segment_name*** clause, or the table will migrate to the *default* segment.

Sharing Space on Segments

When all databases and database objects on a SQL Server share a default pool of space, any table or index might grow to fill the entire pool of space. Similarly, if you place several database objects on a single segment, any of those objects may grow to fill the entire segment. They cannot grow beyond the devices named by that segment to contend for space with objects on other devices, but they will be in contention with each other for space.

Placing Text Pages on a Separate Device

When you create a table with *text* or *image* columns, the data is stored on a separate chain of text pages. A table that contains *text* or *image* columns has an additional entry in *sysindexes* for the text chain, with the name column set to the name of the table preceded by the letter “t” and an *indid* of 255. You can use **sp_placeobject** to store the text chain on a separate device, giving both the table name, and the name of the text chain from *sysindexes*:

```
sp_placeobject textseg, "mytab.tmytab"
```

➤ ***Note***

By default, a chain of text pages is placed on the same segment as its table. After you execute **sp_placeobject**, whatever pages were previously written on the old device remain allocated, but all **new** allocations take place on the new segment.

Information on Storage

To find the names of the database devices on which a particular database resides, use the system procedure **sp_helpdb** with the database name:

```
sp_helpdb pubs2
```

```
name      db_size    owner     dbid created        status
--------- ---------- --------- ---- -------------- --------------
pubs2         2.0 MB sa           5 May 25, 1993   no options set

device_fragments    size         usage            free kbytes
------------------- ------------ ---------------- -----------
pubdev              2.0 MB       data and log             288

device                 segment
---------------------- ----------------------
pubdev                 default
pubdev                 logsegment
pubdev                 system
```

sp_helpdb reports on the size and usage of the devices in use by the named database. If you are using the named database, **sp_helpdb** also reports on the segments in the database, and the devices that are named by the segments.

When **sp_helpdb** is used without arguments, it reports on all the databases on SQL Server:

```
sp_helpdb
```

```
name           db_size  owner dbid created     status
------------- -------- ----- ---- ----------- ------------------
master          3.0 MB sa       1 Jan 01, 1900 no options set
model           2.0 MB sa       3 Jan 01, 1900 no options set
mydata          4.0 MB sa       7 Aug 25, 1993 no options set
pubs2           2.0 MB sa       6 Aug 23, 1993 no options set
sybsecurity    20.0 MB sa       5 Aug 18, 1993 no options set
sybsystemprocs 10.0 MB sa       4 Aug 18, 1993 trunc log on chkpt
tempdb          2.0 MB sa       2 Aug 18, 1993 select into/bulkcopy
```

To get a summary on the amount of storage space used by a database, execute the system procedure **sp_spaceused** in the database:

```
sp_spaceused
```

```
database_name                  database_size
------------------------------ -------------
pubs2                          2.0 MB

reserved      data          index_size      unused
------------- ------------- --------------- --------
1720 KB       536 KB        344 KB          840 KB
```

The columns in the report are:

database_size gives the total amount of space allocated to the database by **create database** or **alter database** commands.

reserved reports the amount of space that has been allocated to all the tables and indexes that have been created in the database. (Space is allocated to database objects inside a database in increments of one extent, or 8 pages, at a time.)

data and *index_size* shows how much space has actually been used by data and indexes.

unused shows the amount of space that has been reserved but not yet used by existing tables and indexes.

Adding the values in the *unused, index_size*, and *data* columns should give you the figure in the *reserved* column. Subtracting *reserved* from *database_size* gives you the amount of unreserved space. This space is available to new objects or to existing objects which grow beyond the space that has already been reserved for them.

Running **sp_spaceused** regularly enables you to monitor the amount of database space available. For example, if the value of *reserved* is close to the total *database_size*, you are close to running out of space for new objects. If *unused* is also quite small, you are close to running out of space for additional data as well.

You can also use **sp_spaceused** with a table name as its parameter, and get a report on the space used by that table, like this:

```
sp_spaceused titles
```

```
name   rowtotal reserved  data    index_size  unused
------ -------- --------- ------- ----------- ------
titles 18       48 KB     6 KB    4 KB        38 KB
```

The rowtotal column in this report may be very different than the results of running **select count(*)** on the table, because **sp_spaceused** computes the value with the built-in function **rowcnt**. This function uses values stored in the allocation pages that record the average number of rows per page. These values aren't updated regularly, however, so they can be very different for tables with a lot of activity. The **update statistics** command, **dbcc checktable** and **dbcc checkdb** update the rows-per-page estimate, so *rowtotal* will be most accurate after one of these commands has been run.

It is a good idea to run **sp_spaceused** regularly on *syslogs*, since the transaction log can grow rapidly if there are frequent database modifications. This is a particular problem if the transaction log is not on a separate device—in which case it will be competing with the rest of the database for space.

You may want to write some of your own queries for additional information about physical storage. For example, to determine the total number of 2K blocks of storage space that exist on SQL Server, you can query *sysdevices*:

```
select sum(high - low)
from sysdevices
where status in (2, 3)
```

```
-------------------
               7168
```

A “2” in the *status* column represents a physical device; a “3” represents a physical device that is also a default device.

4 Managing SQL Server Logins and Database Users

Introduction

This chapter describes methods for managing SQL Server login accounts and database users. Topics covered include:

- Adding new logins to SQL Server
- Creating groups
- Adding users to databases
- Dropping logins, users, and groups
- Locking SQL Server logins
- Changing user information
- Changing user information such as passwords and group membership
- Obtaining information about users, groups, and SQL Server usage

Adding New Users: an Overview

The process of adding new logins to SQL Server, adding users to databases, and granting them permission to use commands and database objects is divided between the System Security Officer, System Administrator, and Database Owner.

The process of adding new users consists of the following steps:

1. A System Security Officer creates a server login account for a new user is created with **sp_addlogin**.
2. A System Administrator or Database Owner adds a user to a database with **sp_adduser**. This command can also give the user an alias or assign the user to a group. Groups are added using **sp_addgroup**. See "Creating Groups: sp_addgroup" on page 4-5 for more information.
3. A System Administrator, Database Owner, or object owner grants the user or group specific permissions on specific commands and database objects. Users or groups can also be granted permission to grant specific permissions on objects to other users or groups. See Chapter 5, "Managing User Permissions" for detailed information on permissions.

The following table summarizes the commands and system procedures used for these tasks.

Command or Procedure	Task	Executed By	Where
sp_addlogin	Create new logins, assign passwords, default databases, default language, and full name,	SSO	any database
sp_addgroup	Create groups	DBO or SA	user database
sp_adduser	Add users to database, assign aliases, assign groups	DBO or SA	user database
grant	Grant groups or users permission on commands and database objects	DBO, SA, or object owner	user database

Table 4-1: System Procedures for Adding Users to SQL Server and Databases

Choosing a Password

Your password is the first line of defense against SQL Server access by unauthorized people. SQL Server passwords must be at least six bytes long and can contain any printable letters, numerals, or symbols. When you create your own password or one for another user, choose one that cannot be guessed. Do not use personal information, names or nicknames of pets or loved ones, or words that appear in the dictionary. The most secure passwords are ones that combine upper- and lowercase characters with numbers, punctuation, and accented characters. Once you select a password, protecting it is your responsibility. Never give anyone your password and never write it down where anyone can see it.

Adding Logins to SQL Server

The system procedure **sp_addlogin** adds new login names to SQL Server. It does not give the user permission to access any user databases; the procedure **sp_adduser** gives users access to specific databases. Only the System Security Officer can execute **sp_addlogin**. Here is its syntax:

```
sp_addlogin login_name, password [, defdb]
   [, deflanguage [, fullname]]]
```

- The *login_name* parameter (the new user's login name) is required. The login name must follow the rules for identifiers and must be unique on SQL Server. It is highly recommended that the SQL Server login name be the same as the user's operating system login name. This makes logging into SQL Server easier, since many client programs use the operating system login name as a default. It also simplifies management of server and operating system login accounts, and makes it easier to correlate usage and audit data generated by SQL Server and by the operating system.

- The *password* parameter, also required, is a password for the new user. The password must be at least six bytes long, and can be any printable letters, numerals, or symbols. A password must be enclosed in quotation marks in **sp_addlogin** if:
  - It includes characters besides A-Z, a-z, 0-9,_, #, valid single- or multi-byte alphabetic characters, or accented alphabetic characters
  - It begins with 0-9

  See "Choosing a Password" on page 4-2 for guidelines on choosing secure passwords. A user can change his or her own password, and a System Security Officer can change any user's password, with **sp_password.**
- *defdb* specifies the name of the new user's default database. This causes the user to start each session on SQL Server in that database, without having to issue the **use** command. A System Administrator can change anyone's default database with **sp_modifylogin**; other users can only change their own.

➤ ***Note***

If you do not specify a user's default database parameter with **sp_addlogin**, the user's default database is *master*. Assign default databases other than *master* to most users, to discourage users from creating database objects in the *master* database.

After you specify the default database, add the user to the default database with **sp_adduser**, so that he or she can log in directly to the default database.

- *deflanguage* specifies the default language in which this user's prompts and messages are displayed. If you omit this parameter, SQL Server's default language is used. The System Administrator can change any user's default language with **sp_modifylogin**; other users can only change their own.

- *fullname* specifies a full name for the user. This can be useful for documentation and identification purposes. If omitted, no full name is added. The System Administrator can change any user's full name with **sp_modifylogin**; other users can only change their own.

The following statement sets up an account for the user "maryd" with the password "100cents," the default database (*master*), the default language, and no full name:

```
sp_addlogin "maryd", "100cents"
```

Quotation marks are not required around character-type parameters to stored procedures as long as they do not contain spaces, punctuation marks or Transact-SQL reserved words. However, it is never wrong to use quotes in **sp_addlogin**.

Once this statement is executed, "maryd" can log in to the SQL Server. She is automatically treated as a guest user in *master*, with limited permissions, unless she has been specifically given access to *master*.

Here's a statement that sets up a login account and a password ("rubaiyat") for user "omar_khayyam," and makes *pubs2* his default database:

```
sp_addlogin omar_khayyam, rubaiyat, pubs2
```

If you wish to specify a full name for a user but use the default database and language, you must specify **null** in place of those two parameters. For example:

```
sp_addlogin omar, rubaiyat, null, null,
    "Omar Khayyam"
```

As an alternative, you can specify a parameter name, in which case you do not have to specify all the parameters. For example:

```
sp_addlogin omar, rubaiyat,
    @fullname = "Omar Khayyam"
```

Effects on System Tables

When you execute **sp_addlogin**, SQL Server adds a row to *master.dbo.syslogins*, assigns a unique server user ID (*suid*) for the new user, and fills in other information. When a user logs in, SQL Server looks in *syslogins* for the name and password the user gives. The *password* column is encrypted with a one-way algorithm and so is not human-readable.

The *suid* column in *syslogins* uniquely identifies the user on SQL Server. A user's *suid* remains the same no matter what database he or she is using. The *suid* 1 is always assigned to the default "sa" account that is created when SQL Server is installed. Other users' server user IDs are small integers assigned consecutively by SQL Server each time **sp_addlogin** is executed.

Permissions Required

Only a System Security Officer can execute **sp_addlogin**.

Creating Groups: *sp_addgroup*

Groups provide a convenient way to grant and revoke permissions to more than one user in a single statement. Groups enable you to provide a collective name to a group of users. They are especially useful if you administer a SQL Server installation that has large numbers of users.

Every user is a member of the group "public," and can be a member of one other group. (Users remain in "public" whether or not they belong to another group.)

It is probably most convenient to create groups before adding users to a database since the **sp_adduser** procedure can assign users to groups as well as adding them to the database. A System Administrator or the Database Owner can create a group at any time with **sp_addgroup**. The syntax for **sp_addgroup** is:

sp_addgroup *groupname*

The group name, a required parameter, must follow the rules for identifiers. For example, to set up the Senior Engineering group, use the following command while using the database to which you wish to add the group:

```
sp_addgroup senioreng
```

The System Administrator can assign or re-assign users to groups with **sp_changegroup**.

Permissions Required

sp_addgroup can be executed by the Database Owner or by a System Administrator.

Effects on System Tables

The **sp_addgroup** system procedure adds a row to *sysusers* in the current database. In other words, each group in a database—as well as each user—has an entry in *sysusers*.

Adding Users to Databases: *sp_adduser*

The procedure **sp_adduser** adds a user to a specific database. The user must already have a SQL Server login. Here is the syntax for **sp_adduser**:

```
sp_adduser login_name [, name_in_db [, groupname]]
```

- *login_name* is the login name of an existing user. It is required.
- *name_in_db* allows you to specify a name different from the login name by which the user is to be known inside the database. It is optional.

  This feature is supplied to accommodate users' preferences. For example, if there are five SQL Server users named Mary, all of them must have different login names. Mary Doe might log in as "maryd," Mary Jones as "maryj," and so on. However, if the Marys don't use the same databases, each might prefer to be known simply as "mary" inside a particular database.

  If no *name_in_db* parameter is given, the name inside the database is the same as *login_name*.

  Be careful not to confuse this capability with the alias mechanism described in ''Using Aliases in Databases'' on page 4-16, which maps the identity and privileges of one user to another.

- *groupname*, also optional, is the name of an existing group in the database. If you do not specify a group name, the user is made a member of the default group "public". Users remain in "public" even if they are a member of another group. See ''Changing a User's Group Membership: sp_changegroup'' on page 4-15 for information on modifying a user's group membership.

Here's a command that the owner of the *pubs2* database could give to allow "maryh" of the (already existing) engineering group to access *pubs2*:

```
sp_adduser maryh, mary, eng
```

Here's how to give "maryd" access to *pubs2*, with her name in the database the same as her login name, and making her a member of the "public" group:

```
sp_adduser maryd
```

Here's how to add a user and put the user in a group, but without different names, using **null** in place of a new user name:

```
sp_adduser maryj, null, eng
```

Users who have access to a database still need permission to do things inside it: to read data, modify data, and use certain commands. These permissions are set up by the owners of the database objects or user databases with the **grant** and **revoke** commands. They are discussed in Chapter 5, ''Managing User Permissions''.

Effects on System Tables

The **sp_adduser** system procedure adds a row to the *sysusers* system table in the current database. Once a user has an entry in the *sysusers* table of a database, he or she:

- Can issue the **use** command and access that database,
- Will use that database by default, if the default database parameter was issued as part of **sp_addlogin**, or
- Can use **sp_modifylogin** to make that database the default.

Permissions Required

sp_adduser can be executed by the Database Owner and System Administrator.

Adding a "guest" User

Creating a user named "guest" in a database enables any user with a SQL Server account to access the database as a guest user. If a user issues the "**use *database_name***" command, and his or her name is not found in the database's *sysusers* table, SQL Server looks for a guest user. If there is one, the user is allowed to access the database, with the permissions of the guest user.

The Database Owner can add a guest entry to the *sysusers* table of any user database with the system procedure **sp_adduser**, like this:

```
sp_adduser guest
```

The guest user can be removed with **sp_dropuser**, as discussed in ‘‘Dropping Database Users: sp_dropuser’’ on page 4-11.

If you drop the guest user from the *master* database, server users who have not yet been added to any databases will be unable to log into SQL Server.

“guest” User Permissions

When you first add the username “guest” to a database, “guest” inherits the privileges of “public”. The Database Owner and the owners of database objects can change these permissions as they wish, using **grant** and **revoke**, to make the privileges of “guest” either more restrictive or less restrictive than those of “public.” See Chapter 5, ‘‘Managing User Permissions’’, for a description of the privileges “public” has.

When you install SQL Server, *master..sysusers* contains a guest entry. The installation script for the *pubs2* database also contains a guest entry for its *sysusers* table.

The “guest” User in User Databases

In user databases, the Database Owner is responsible for setting up any guest mechanisms that are needed. Adding a guest user to a user database allows an owner to permit all SQL Server users to use that database without having to use **sp_adduser** to explicitly name each one as a database user.

You can use the guest mechanism to restrict access to database objects while allowing access to the database.

For example, the owner of the *titles* table could grant **select** permission on *bugs* to all database users except “guest” by executing the following three commands:

```
grant select on titles to public

sp_adduser guest

revoke all on titles from guest
```

If you wish to grant permissions to “public”, but do not wish to grant these permissions to “guest”, be sure to issue a **revoke** command after granting permission to “public”. The group “public” includes all users.

The "guest" User in **pubs2**

The "guest" user entry in *pubs2*, the sample database, allows new SQL Server users to follow the examples in the *Transact-SQL User's Guide* and to engage in a certain amount of experimentation. The guest in *pubs2* is given a wide range of privileges:

- **select** permission and data modification permission on all of the user tables
- **execute** permission on all of the procedures
- **create table, create view, create rule, create default** and **create procedure** permission

Visitor Accounts on SQL Server

As another method of accommodating visiting users on SQL Server, the System Security Officer can use **sp_addlogin** to enter a row in *master..syslogins* with a login name and password that visiting users are instructed to use (for example, "visitor"). Typically, such users are granted very restricted permissions. A default database may be assigned. Allowing more than one person to use the visitor account does not provide good individual accountability, however, since the actions of individual users of the visitor account cannot be audited.

Be careful not to confuse such visitor accounts with the guest user mechanism described earlier.

Adding Remote Users

You can allow users on another SQL Server to execute stored procedures on your server by enabling remote access. Working with a System Administrator of the remote server, you can also allow users of your server to execute remote procedure calls to the remote server. To enable remote procedure calls, both the local and the remote server must be configured, following several steps. See Chapter 15, ‘‘Managing Remote Servers’’, for information on setting up remote servers and adding remote users.

Granting Permissions to Database Users

The final step in adding database users is granting them permission to use commands and database objects. See Chapter 5, ‘‘Managing User Permissions’’.

Dropping Logins, Users, and Groups

The following system procedures allow a System Administrator or Database Owner to drop logins, users and groups.

System Procedure	Task	Executed By	Where
sp_droplogin	Drop user from SQL Server	SA	*master*
sp_dropuser	Drop user from database	DBO or SA	user database
sp_dropgroup	Drop group from database	DBO or SA	user database

Table 4-2: Dropping Logins, Users, and Groups

Dropping Logins: *sp_droplogin*

The system procedure **sp_droplogin** denies a user access to SQL Server. It may be easier to lock logins rather than delete them, for these reasons:

- You cannot drop a login who is a user in any database, and you cannot drop a user from a database if the user owns any objects in that database or has granted any permissions on objects to other users.
- SQL Server may reuse the dropped login account's server user ID (*suid*) when the next login account is created. This only occurs when the dropped login holds the highest *suid* in *syslogins*, but could compromise accountability if execution of **sp_droplogin** is not being audited.
- You cannot drop the last remaining System Security Officer's or System Administrator's login account.

See "Locking SQL Server Logins: sp_locklogin" on page 4-11 for information on locking logins.

Here is the syntax for **sp_droplogin**:

```
sp_droplogin login_name
```

Dropping Database Users: *sp_dropuser*

The system procedure **sp_dropuser** denies a SQL Server user access to the database in which it is executed. (If there is a "guest" user defined in that database, the user can still access it as "guest".)

Here is the syntax for dropping a user from a database:

sp_dropuser *name_in_db*

name_in_db is usually the login name, unless another name has been assigned.

sp_dropuser can be executed only by the Database Owner or a System Administrator.

➤ ***Note***

You cannot drop a user who owns objects. Since there is no command to transfer ownership of objects, you must drop objects owned by a user before you drop the user with **sp_dropuser**. If you wish to deny access to a user who owns objects, use **sp_locklogin** to lock his or her account.

You also cannot drop a user who has granted permissions to other users. **sp_dropuser** displays an informational message when this happens. Use **revoke with cascade** to revoke permissions from all users who were granted permissions by the user to be dropped, then drop the user. You must then re-grant permissions to the users, if appropriate.

Dropping Groups: *sp_dropgroup*

To drop a group, use **sp_dropgroup** and give the group name:

sp_dropgroup *groupname*

You cannot drop a group that has members. If you try to do so, the error report displays a list of the members of the group you are attempting to drop. To remove users from a group, execute **sp_changegroup**, discussed in "Changing a User's Group Membership: sp_changegroup" on page 4-15.

Locking SQL Server Logins: *sp_locklogin*

Locking a SQL Server login account prevents that user from logging in. You may prefer to lock accounts instead of dropping them for reasons discussed in ‘‘Dropping Logins: sp_droplogin’’ on page 4-10. The **sp_locklogin** system procedure locks and unlocks accounts, or displays a list of locked accounts.

The syntax is:

sp_locklogin [*login_name*, "{lock | unlock}"]

- *login_name* is the name of the account to be locked or unlocked. It must be an existing, valid account.
- **lock** | **unlock** specifies whether the account is to be locked or unlocked.

Using **sp_locklogin** with no parameters displays a list of all locked logins.

You can lock an account that is currently logged in. You will receive a warning that the account has been locked, but the user is not locked out of the account until he or she logs out. You can lock the account of a Database Owner, and a locked account can own objects in databases. In addition, you can use **sp_changedbowner** to specify a locked account as the owner of a database.

SQL Server ensures that there is always at least one unlocked System Security Officer’s account and one unlocked System Administrator’s account.

Permissions Required

Only System Administrators and System Security Officers can execute **sp_locklogin**.

Changing User Information

Additional system procedures allow you to change any of the user information added with the commands discussed earlier in this chapter. For example, **sp_modifylogin** and **sp_password** change default databases and passwords for SQL Server users.

The system procedures described in Table 4-3 change aspects of users' logins and database usage.

System Procedure	Task	Executed By	Where
sp_password	Change another user's password	SSO	any database
	Change own password	user	
sp_changegroup	Change group assignment of a user	SA, DBO	user database
sp_modifylogin	Change a login account's default database, default language, or full name	SA	any database
	Change own default database, default language, or full name	user	

Table 4-3: System Procedures for Changing User Information

Changing Passwords: *sp_password*

Your site may choose to use the **password expiration interval** configuration variable to establish a password expiration interval to force all SQL Server users to change their passwords on a regular basis. (See "password expiration interval" on page 12-37 for information.) Even if you do not use **password expiration interval**, for security reasons it is important that users change their passwords from time to time.

The column *pwdate* in *syslogins* records the date of the last password change. This query selects all login names whose passwords have not been changed since January 30th, 1994:

```
select name, pwdate
from syslogins
where pwdate < "Jan 30 1994"
```

A user can change passwords at any time with the system procedure **sp_password.** The System Security Officer can use this system procedure to change any other user's password. The syntax is:

sp_password *caller_passwd*, *new_passwd* [, *login_name*]

- *caller_passwd* is the password of the login account that is currently executing **sp_password**. When you are changing your own password, this is your old password. When a System Security Officer uses **sp_password** to change another user's password, *caller_passwd* is the password of the System Security Officer.

- *new_passwd* is the new password for the user executing **sp_password**, or for the user indicated by *login_name*. The password must be at least six bytes long, and can be any printable letters, numerals, or symbols. A password must be enclosed in quotation marks if:
  - It includes characters other than A-Z, a-z, 0-9, _, #, valid single- or multi-byte alphabetic characters, or accented alphabetic characters
  - It begins with 0-9
- See ‘‘Choosing a Password’’ on page 4-2 for guidelines on selecting a password.
- *login_name* may only be used by a System Security Officer to change another user’s password.

For example, a user can change her password from “3blindmice” to “2mediumhot” with this command:

```
sp_password "3blindmice", "2mediumhot"
```

These passwords are enclosed in quotes because they begin with numerals.

In the following example, the System Security Officer whose password is “2tomato” changes Victoria’s password to “sesame1”:

```
sp_password "2tomato", sesame1, victoria
```

Null Passwords

You may not assign a null password. However, when SQL Server is installed, the default “sa” account has a null password. The following example changes a null password to a valid one:

```
sp_password null, "16tons"
```

Note that “null” is not enclosed in quotes.

Permissions Required

All users can use **sp_password** to change their own passwords. Only a System Security Officer can use it to change another user’s password.

Changing User Defaults with *sp_modifylogin*

The **sp_modifylogin** system procedure allows a user to change his or her default database, default language, or full name at any time. The

System Administrator can use **sp_modifylogin** to change these for any user. The syntax is:

```
sp_modifylogin login_name, option, value
```

- *login_name* is the name of the user whose account you are modifying.
- *option* specifies the option that you are changing. The options are listed in Table 4-4, and are described in more detail in the sections following.

Option	Definition
defdb	The "home" database to which the user is connected when he or she logs in.
deflanguage	The official name of the user's default language, as stored in *master..syslanguages*.
fullname	The user's full name.

Table 4-4: Options for sp_modifylogin

- *value* is the new value for the specified option.

After you execute **sp_modifylogin** to change the default database, the user is connected to the new default database the next time he or she logs in. However, **sp_modifylogin** does not automatically give the user access to the database. Unless the Database Owner has set up access with **sp_adduser**, **sp_addalias**, or with a guest user mechanism, the user is connected to *master* even after his or her default database has been changed.

Examples

This example changes the default database for "anna" to *pubs2*:

```
sp_modifylogin anna, defdb, pubs2
```

This example changes the default language for "claire" to French:

```
sp_modifylogin claire, deflanguage, french
```

This example changes the full name for "clemens" to "Samuel Clemens."

```
sp_modifylogin clemens, fullname, "Samuel Clemens"
```

Changing a User's Group Membership: *sp_changegroup*

The System Administrator can use the system procedure **sp_changegroup** to change an existing user's group affiliation. At any one time, each user can be a member of only one group other than "public," of which all users are always members. When new users are added to the database, they are assigned to the group "public" as well as any group specified with the **sp_adduser** procedure.

Before you execute **sp_changegroup**:

- The group must exist. (Use **sp_addgroup** to create a group.)
- The user must have access to the current database (must be listed in *sysusers*).

The syntax for **sp_changegroup** is:

```
sp_changegroup grpname, name_in_db
```

For example, here's how you'd change Jim from his current group to the group "manage":

```
sp_changegroup manage, jim
```

To remove a user from a group without assigning the user to another group, you must change the group affiliation to "public":

```
sp_changegroup "public", jim
```

The name "public" must be in quotes because it is a reserved word. All users are always members of "public." This command reduces Jim's group affiliation to "public" only.

When a user changes from one group to another, the user loses all permissions that he or she had as a result of belonging to the old group. That user gains the permissions that have been granted to the new group.

The assignment of users into groups can be changed at any time.

Using Aliases in Databases

The alias mechanism allows you to treat more than one person as the same user inside a database, so that they all have the same privileges. It is often used so that several users can assume the role of Database Owner. The alias mechanism can also be used to set up a collective user identity, within which the identities of individual users can be traced by auditing their activities.

For example, say several vice presidents want to use a database with identical privileges and ownerships. One way to accomplish this is to add a login named "vp" to SQL Server and the database; each vice president logs in as "vp". The problem with this method is that there is no way to tell the individual users apart. The other approach is to alias all the vice presidents, each of whom has his or her own SQL Server account, to the database user name “vp”.

The following system procedures are used to manage aliases:

System Procedure	Task	Executed By	Where
sp_addalias	Add an alias for a user	DBO or SA	user database
sp_dropalias	Drop an alias	DBO or SA	user database

Table 4-5: System Procedures for Managing Aliases

Adding Aliases: *sp_addalias*

Here’s the syntax for **sp_addalias**:

```
sp_addalias login_name, name_in_db
```

- *login_name* is the name of the user who wants an alternate identity in the current database. This user must have an account on SQL Server but cannot be a user in the current database.
- *name_in_db* is the name of the database user to whom the first user wishes to be linked. This name must exist in both *master..syslogins* and in *sysusers* in the current database. Both parameters are required.

Executing **sp_addalias** maps the user with the specified login name to the user with the specified *name_in_db*. It does this by adding a row to the system table *sysalternates*.

When a user tries to **use** a database, SQL Server checks for the user’s server user ID number (*suid*) in *sysusers*. If it is not found, SQL Server then checks *sysalternates*. If the user’s *suid* is found there, mapped to a database user’s *suid*, the first user is treated as the second user while using the database.

As an example, suppose that Mary owns the database. She wishes to allow both Jane and Sarah to use the database as if they were its owner. Jane and Sarah have logins on SQL Server but are not authorized to use Mary’s database. Mary executes these commands:

```
sp_addalias jane, dbo
exec sp_addalias sarah, dbo
```

Now both Jane and Sarah can access Mary’s database, and be recognized as its owner.

Dropping Aliases: *sp_dropalias*

The system procedure **sp_dropalias** drops the mapping of an alternate suid to a user ID, deleting the relevant row from *sysalternates*. Its syntax is:

sp_dropalias *login_name*

login_name is the name of the user who was mapped to another user. Once a user's alias is dropped, the user no longer has access to the database, unless his or her name is then added to *sysusers* (with **sp_adduser**) or the database has a "guest" user in *sysusers*.

Getting Information on Aliases

To display information about aliases, use the **sp_helpuser** procedure. Here's an example:

```
sp_helpuser dbo
```

```
Users_name     ID_in_db    Group_name   Login_name   Default_db
----------     --------    ----------   ----------   ---------
dbo            1           public       sa           master

(1 row affected)
Users aliased to user.
Login_name
----------------------
andy
christa
howard
linda

(4 rows affected)
```

Getting Information on Users

The following procedures allow users to obtain information about users, groups and current SQL Server usage.

Procedure	Function
sp_who	Reports on current SQL Server users and processes
sp_displaylogin	Displays information about login accounts
sp_helpuser	Reports on users and aliases in a database
sp_helpgroup	Reports on groups within a database

Table 4-6: System Procedures that Report on SQL Server Users and Groups

Getting Reports on SQL Server Users and Processes: *sp_who*

The system procedure **sp_who** reports information on current users and processes on SQL Server. Its syntax is:

sp_who [*login_name* | "*spid*"]

login_name is the user's SQL Server login name. If you give a login name, **sp_who** reports information on processes being run by the specified user.

spid is the number of a specific process. Enclose it in quotes because a character type argument is expected.

For each process being run, **sp_who** reports the server process ID, its status, the login name of the process user, the name of the host computer, the server process ID of a process that's blocking this one (if any), the name of the database, and the command being run.

If you do not give a login name, **sp_who** reports on processes being run by all users.

Here's an example of the results of executing **sp_who** without a parameter:

```
spid   status   loginame hostname blk dbname cmd
------ -------- -------- -------- --- ------ ---------------
     1 running  sa       sunbird  0   pubs2  SELECT
     2 sleeping NULL              0   master NETWORK HANDLER
     3 sleeping NULL              0   master MIRROR HANDLER
     4 sleeping NULL              0   master AUDIT PROCESS
     5 sleeping NULL              0   master CHECKPOINT SLEEP

(5 rows affected, return status = 0)
```

For all system processes, **sp_who** reports NULL for the *login_name*.

Getting Information about Login Accounts: *sp_displaylogin*

sp_displaylogin displays information about a specified login account. The syntax is:

sp_displaylogin [*login_name*]

login_name is the user login account about which you want information. If you wish to display information about your own login, you do not need to specify *login_name*.

Only a System Administrator or System Security Officer can use **sp_displaylogin** (with the *login_name* parameter) to get information about other logins.

sp_displaylogin displays the following information about any login account:

Server user ID
Login name
Full name
Any roles granted
Whether the account is locked
Date that the password was last changed

sp_displaylogin displays all roles that have been granted to you, so that even if you have made a role inactive with the **set** command, it is displayed.

Getting Information about Database Users: *sp_helpuser*

The system procedure **sp_helpuser** reports information on authorized users of the current database. Its syntax is:

sp_helpuser [*name_in_db*]

The parameter, the database name of a user, is optional. If you give a user's name, **sp_helpuser** reports information on that user. If you don't give a name, it reports on all users.

The procedure reports the user's name in the database, the user ID, the user's login name, and the group name. Here's an example of the results of executing **sp_helpuser** without a parameter in the database *pubs2*:

```
sp_helpuser
```

```
Users_name  ID_in_db    Group_name    Login_name  Default_db
----------  --------    ----------    ----------  ----------
dbo                1    public        sa          master
marcy              4    public        marcy       pubs2
sandy              3    public        sandy       pubs2
judy               5    public        judy        pubs2
linda              6    public        linda       master
anne               2    public        anne        pubs2
jim                7    senioreng     jim         master

(7 rows affected)
```

Finding User Names and IDs

To find a user's server user ID or login name, use the system functions **suser_id** and **suser_name**.

Function	Argument	Result
suser_id	([“*server_user_name*”])	server user ID
suser_name	([*server_user_ID*])	server user name (login name)

Table 4-7: System Functions suser_id and suser_name

The arguments for these system functions are optional. If you don't give one, SQL Server displays information about the current user. For example, here we give names and numbers from the **sp_helpuser** example above:

```
select suser_name(3), suser_id("sandy")

--------------- ------
sandy                3
```

Here the System Administrator issues the commands without arguments:

```
select suser_name(), suser_id()
```

```
--------------- ------
sa                   1
```

To find a user's ID number or name inside a database, use the system functions **user_id** and **user_name**.

Function	Argument	Result
user_id	([“*db_user_name*”])	user ID
user_name	([*db_user_ID*])	user name

Table 4-8: System Functions user_id and user_name

The arguments for these system functions are optional. If you don't give one, SQL Server displays information about the current user. For example:

```
select user_name(10)
select user_name( )
select user_id("joe")
```

Getting Information about Usage: Chargeback Accounting

When a user logs into SQL Server, the server begins accumulating CPU and I/O usage for that user. Using this information, SQL Server can report total usage for an individual or for all users. Information for each user is kept in the *syslogins* system table in the *master* database.

A System Administrator can configure the frequency with which usage statistics are updated by setting the configuration variables **cpu flush** and **i/o flush**. Whenever the user logs out of a SQL Server session or accumulates more CPU or I/O usage than the configured **cpu flush** or **i/o flush** value, the new totals are written to *syslogins*.

System Procedures for Reporting Current Usage Statistics

The System Administrator can use either of the system procedures **sp_reportstats** or **sp_clearstats** to get current total usage for individuals or for all users on a SQL Server. To get one user's totals, use the SQL Server login as a parameter; to get the total for all users, use **sp_clearstats** or **sp_reportstats** with no parameters.

Using sp_reportstats

sp_reportstats displays a report of current accounting totals for SQL Server users. It reports total CPU and total I/O, as well as the percentage of those resources used. It does not record statistics for processes with an *suid* of 1 (the “sa” login), checkpoint, network, and mirror handlers.

Using sp_clearstats

SQL Server continues to accumulate CPU and I/O statistics until you clear the totals from *syslogins* by running **sp_clearstats**. **sp_clearstats** initiates a new accounting interval for SQL Server users. It executes **sp_reportstats** to print out statistics for the previous period and clears *syslogins* of the accumulated CPU and I/O statistics.

Choose the length of your accounting interval by deciding how you wish to use the statistics at your site. To do monthly cross-department charging for percentage of SQL Server CPU and I/O usage, for example, the System Administrator would run **sp_clearstats** once a month.

For detailed information on these stored procedures, see Volume 2 of the *SQL Server Reference Manual*.

Configuration Variables for Chargeback Accounting

Two configuration variables allow a System Administrator to decide how often accounting statistics are added to *syslogins*:

- **cpu flush** specifies how many machine clock ticks to accumulate before adding to *syslogins*. To find out how many microseconds a tick is on your system, run this query while logged into SQL Server:

  ```
  select @@timeticks
  ```

- **i/o flush** specifies how many read or write I/Os to accumulate before flushing to *syslogins*. I/O and CPU statistics for an individual user are written out whenever the user exits a SQL Server session or accumulates more CPU or I/O usage than these values.

The minimum value allowed for either variable is 1; the maximum value allowed is 2147483647. The default value for **cpu flush** is 300 and the default value for **i/o flush**, 1000.

To set these values, use the **sp_configure** command:

```
sp_configure "cpu flush", 600
```

or:

```
sp_configure "i/o flush", 2000
```

Next, run the **reconfigure** command to change the configuration values.

5 Managing User Permissions

Introduction

The SQL commands **grant** and **revoke** control SQL Server's command and object protection system. You can give various kinds of permissions to users, groups, and roles with the **grant** command, and rescind them with the **revoke** command. **grant** and **revoke** are used to specify which users can issue which commands and perform which operations on which tables, views, or columns. The terms **granting, assigning privileges,** and **permissions** are used to describe the use of the **grant** and **revoke** commands.

Some commands can be used at any time by any user, with no permission required. Others can be used only by users of certain status (for example, only by a System Administrator) and are not transferable.

The ability to assign permissions for the commands that can be granted and revoked is determined by each user's status (as System Administrator, Database Owner, or database object owner), and by whether or not a particular user has been granted a permission with the option to grant that permission to other users.

To supplement and complement the **grant** and **revoke** commands, you can use views and stored procedures as a security mechanism. This is discussed in "Permissions on Views and Stored Procedures" on page 5-20.

Permission Summary

System Administrators operate outside of the command and object protection system, and have all permissions on commands and objects at all times.

Database Owners do not automatically receive permissions on objects owned by other users, but they can impersonate other users in their database at any time with the **setuser** command, temporarily assuming their permissions. Also, a Database Owner can permanently acquire permission on a given object by assuming the identity of the object owner with the **setuser** command and then issuing the appropriate **grant** or **revoke** statements. The **setuser** command is discussed in more detail later in this chapter.

For permissions that default to "public," no permission is required—that is, no **grant** or **revoke** statements need ever be written.

Table 5-1 summarizes the SQL Server protection system. The type of user listed as the one to whom the command defaults is the "lowest" level of user to which the permission is automatically granted. This user can **grant** the permission to other users or **revoke** it from other users, if it is transferable.

This table does not include System Security Officers. The System Security Officer does not have any special permissions on commands and objects, only on certain system procedures.

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 table				•			•	
begin transaction					•			•
checkpoint			•				•	
commit transaction					•			•
create database	•					•		
create default			•			•		
create index				•			•	
create procedure			•			•		
create rule			•			•		
create table			•		(2)	• (2)		
create trigger				•			•	
create view			•			•		
dbcc			•				•	

(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 always restricted
N/A means use of the command is never restricted

Table 5-1: Command and Object Permissions

Command and Object Permissions								
Statement	**Defaults To**					**Can Be Granted/Revoked**		
	System Admin.	**Operator**	**Database Owner**	**Object Owner**	**Public**	**Yes**	**No**	**N/A**
delete				• (3)		•		
disk init	•						•	
disk mirror	•						•	
disk refit	•						•	
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)		
reconfigure	•						•	
references				•		•		
revoke on object				•			•	
revoke command			•				•	

(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 always restricted
N/A means use of the command is never restricted

Table 5-1: Command and Object Permissions (continued)

Command and Object Permissions								
Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
rollback transaction					•			•
save transaction					•			•
select				• (3)		•		
set					•			•
setuser			•				•	
shutdown	•						•	
truncate table				•			•	
update				• (3)		•		
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 always restricted **N/A** means use of the command is never restricted				

Table 5-1: Command and Object Permissions (continued)

Types of SQL Server Users and Their Privileges

SQL Server's command and object permission system recognizes these types of users:

- System Administrators
- System Security Officers
- Operators
- Owners of databases
- Owners of database objects
- Other users (also known as "public")

Privileges of System Administrators

System Administrators are special users who handle tasks not specific to applications. SQL Server recognizes a System Administrator as a super-user who works outside SQL Server's command and object protection system.

The role of System Administrator is usually granted to individual SQL Server logins. This provides a high degree of individual accountability because all actions taken by that user can be traced to his or her individual server user ID. If the server administration tasks at your site are few enough that they are performed by a single individual, you may instead choose to use the "sa" account that is installed with SQL Server. At installation, "sa" has permission to assume the System Administrator, System Security Officer, and Operator roles. Any user who knows the "sa" password can log into that account and assume the System Administrator role.

The fact that a System Administrator operates outside the protection system serves as a safety precaution. For example, if the Database Owner accidentally deletes all the entries in the *sysusers* table, the System Administrator can fix things up (provided, of course, you are making regular backups!).

There are several commands that only a System Administrator can issue—they cannot be granted to any other user. They include: **reconfigure**, **disk init**, **disk refit**, **disk reinit**, the disk mirroring commands, **shutdown** and **kill**.

Permissions for Creating Databases

Only a System Administrator can grant permission to use the **create database** command. The **grant** command for permission on **create database** must be issued from *master*. In many installations, the System Administrator maintains a monopoly on **create database** permission in order to centralize control of database placement and database device space allocation.

In such situations, a System Administrator creates new databases on behalf of other users, and then transfers ownership to the appropriate user. To create a database that is to be owned by another user, the System Administrator issues the **create database** command, switches to the new database with the **use** command, and then executes the system procedure **sp_changedbowner**. System Administrators can, however, grant permission to create databases if that is desired.

Privileges of System Security Officers

System Security Officers perform security-sensitive tasks in SQL Server. These tasks include managing logins (including login aliases and group set-up), granting roles, and changing passwords. System Security Officers do not have special permissions on database objects.

System Security Officers can repair any damage inadvertently done to the protection system by a user. For example, if the Database Owner forgets his or her password, a System Security Officer can change the password to allow the Database Owner to log in.

Privileges of Operators

Users who have been granted the Operator role can back up and restore databases on a server-wide basis without having to be the owner of each one. The Operator role allows a user to use these commands on any database:

dump database
dump transaction
load database
load transaction

Database Owners also have dump and load permissions on the databases they own. System Administrators can dump and load all databases.

Privileges of Database Owners

Database owners and System Administrators are the only users who can grant command permissions to other users.

The owner of a database has full privileges to do anything inside that database, and must explicitly **grant** permissions to other users.

Here are the permissions that are automatically granted to the owner of a database, and that cannot be transferred to others (except that the dump and load commands are granted to users with the Operator role):

checkpoint
dbcc
setuser
dump database
dump transaction

load database
load transaction
drop database
grant and **revoke** command permissions

Database owners can **grant** permissions on the following commands to other users:

create table
create default
create rule
create procedure
create view
grant and **revoke** permissions on system tables
grant and **revoke select, insert, delete, update, references,** and **execute** permissions on database objects

Permissions on System Tables

The installation script supplied by SQL Server sets permissions on the system tables in a user database so that all database users can read them.

However, the default situation is that no users—including Database Owners—can modify the system tables directly. Instead, the system stored procedures supplied with SQL Server modify the system tables. This helps guarantee integrity.

A System Administrator can change this situation so that *ad hoc* modifications to the system tables are allowed with the system procedure **sp_configure** and the **reconfigure** command. See "allow updates" on page 12-26.

Permissions on System Procedures

Since system procedures are stored in the *sybsystemprocs* database, their permissions are also set there.

Security-related system procedures can only be run by System Security Officers. Certain other system procedures can only be run by System Administrators.

Some of the system procedures can be run only by Database Owners. These procedures make sure that the user executing the procedure is the owner of the database from which they are being executed.

Other system procedures (for example, all the **sp_help** procedures) can be executed by any user who has been granted permission—but this permission must be granted in *sybsystemprocs*. In other words, a user must have permission to execute a system procedure in all databases, or in none of them.

Users not listed in *sybsystemprocs..sysusers* are treated as "guest" in *sybsystemprocs*, and are automatically granted permission on many of the system procedures. In order to deny a user permission on a system procedure, the System Administrator must add him or her to *sybsystemprocs..sysusers* and issue a **revoke** statement that applies to that procedure. The owner of a user database cannot directly control permissions on the system procedures within his or her own database.

The setuser Command

A Database Owner can use the **setuser** command to "impersonate" another user's identity and permissions status in the current database. A Database Owner may find it convenient to use **setuser** in order to access an object owned by another user, to grant permissions on an object owned by another user, to create an object that will be owned by another user, or to temporarily take on the privileges of another user for some other reason.

setuser permission defaults to the Database Owner and cannot be transferred. The user being impersonated must be an authorized user of the database.

System Administrators can use **setuser** in order to create objects that will be owned by another user. However, since System Administrators operate outside the permissions system, they cannot use **setuser** to acquire another user's permissions. No matter what **setuser** commands have been issued, System Administrators always retain permission to do everything.

The syntax of the **setuser** command is:

setuser ["*user_name*"]

The user being impersonated must be authorized to use the current database. To re-establish your original identity, give the **setuser** command with no name after it.

Here's how the Database Owner would grant Joe permission to read the *authors* table, which is owned by Mary:

```
setuser "mary"

grant select on authors to joe

setuser     /*re-establishes original identity*/
```

When the Database Owner uses the **setuser** command, SQL Server checks the permissions of the user being impersonated.

The **setuser** command remains in effect until another **setuser** command is given, or until the current database is changed with the **use** command.

Changing Database Ownership

Use the system procedure **sp_changedbowner** to change the ownership of a database. Often, System Administrators create the user databases, then give ownership to another user after some of the initial work is complete. Only the System Administrator can execute **sp_changedbowner.**

It is a good idea to transfer ownership before the user has been added to the database, and before the user has begun creating objects in the database. The new owner must already have a login name on SQL Server, but cannot be a user of the database, or have an alias in the database. You may have to use **sp_dropuser** or **sp_dropalias** before you can change a database's ownership, and you may have to drop objects before you can drop the user.

Issue **sp_changedbowner** in the database whose ownership will be changed. The syntax is:

sp_changedbowner *login_name* [, true]

This example makes the user "albert" the owner of the current database and drops aliases of users who could act as the old "dbo":

```
sp_changedbowner albert
```

To transfer aliases and their permissions to the new "dbo," add the second parameter with the value "true" or "TRUE".

➤ ***Note***

You cannot change the ownership of the *master* database and should not change the ownership of any other system databases.

Privileges of Database Object Owners

A user who creates a database object (a table, view, or stored procedure) owns the objects, and is automatically granted all object permissions on it. Users other than the object owner, including the owner of the database, are automatically denied all permissions on that object, unless they are explicitly granted by either the owner or a user who has **grant** permission on that object.

As an example, say that Mary is the owner of the *pubs2* database, and has granted Joe permission to create tables in it. Now Joe creates the table *new_authors*; he is the owner of this database object.

Initially, object permissions on *new_authors* belong to Joe and Joe alone. Joe can grant or revoke object permissions for this table to other users, including Mary and the Database Owner. (However, the Database Owner can use the **setuser** command as described earlier to impersonate Joe.)

These are the command permissions that default to the owner of a table, and that cannot be transferred to other users:

alter table
drop table
create index
create trigger
truncate table
update statistics

Permission to use the **grant** and **revoke** commands to grant specific users permission to use the **select, insert, update, delete, references,** and **execute** commands on specific database objects can be transferred, using **grant with grant option**.

Permission to **drop** an object—a table, view, index, stored procedure, rule, or default—defaults to its owner and cannot be transferred.

Privileges of Other Database Users

At the bottom of the hierarchy is the "public"—other database users. Permissions are granted to or revoked from them by object owners, Database Owners, users who were granted permissions **with grant option**, and / or a System Administrator. These users are specified by user name, by group name, or by the keyword **public**.

Command and Object Permissions

There are two categories of permissions. Permissions for the commands **select**, **update**, **insert**, **delete**, **references**, and **execute** are called **object permissions** because these commands always apply to database objects (in the current database).

The objects to which the object permissions can apply are:

Command	Objects
select	table, view, columns
update	table, view, columns
insert	table, view
delete	table, view
references	table
execute	stored procedure

Table 5-2: Object Permissions

Object permissions default to System Administrators and the object owner, and can be granted to other users.

Permissions for other commands are called **command permissions** because they are not object specific. They can be granted only by a System Administrator or a Database Owner. These commands are:

Commands	Restrictions
create database	Can be granted only by a System Administrator, and only to users in the *master* database
create default	
create procedure	
create rule	
create table	
create view	

Table 5-3: Commands on Which Permissions May Be Granted

Each database has its own independent protection system: having permission to use a certain command in one database has no effect in other databases.

If you try to use a command or database object for which you have not been assigned permission, SQL Server displays an error message.

Granting and Revoking Permissions

Command permissions and object permissions are given or removed with the **grant** and **revoke** commands.

grant and *revoke* Syntax

The syntax for command permissions differs slightly from the syntax for object permissions. Here are the syntax statements for command permissions:

```
grant {all [privileges] | command_list}
   to {public | name_list | role_name}
```

```
revoke {all [privileges] | command_list}
   from {public | name_list | role_name}
```

The command list can include the following commands in any combination: **create database** (can be granted only by a System Administrator and in *master*), **create default**, **create procedure**, **create rule**, **create table**, and **create view**. If you include more than one command, separate them with commas.

Only a System Administrator or the Database Owner can use the keyword **all**. When used by a System Administrator, **grant all** assigns all **create** permissions. When the Database Owner uses **grant all**, SQL Server grants all **create** permissions except **create database**, and prints an informational message.

Here are the syntax statements for granting and revoking object permissions (permission to use tables, views, columns, and stored procedures):

```
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]
```

```
revoke {all [privileges] | permission_list}
   on {table_name [(column_list)] |
       view_name [(column_list)] |
       stored_procedure_name}
   from {public | name_list | role_name}
        [cascade]
```

- If you are granting or revoking permissions on a table or view, without specifying any columns, the privilege list can consist of any combination of **select**, **insert**, **delete**, **references**, and **update**.

➤ ***Note***

insert and **delete** permissions do not apply to columns, so you cannot include them in a privilege list (or use the keyword **all**) if you specify a column list.

- When you grant or revoke permissions on columns, the privilege list can include only **select**, **references**, and **update**. If you want a user to be able to use a "**select** *" statement, you must grant her or him **select** permission on the table, or on **all** the columns in a table.

  You can specify columns either in the *permission_list*, or in the *column_list*, but not both.

- When you grant permissions on stored procedures, the privilege list must consist of **execute** or **all**, which is a synonym for **execute**.
- When you have **set ansi_permissions on** additional permissions are required for **update** and **delete** statements. The following table summarizes these requirements:

	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 *And...* • **select** permission on all columns appearing in **where** clause • **select** permission on all columns right side of **set** clause
delete	• **delete** permission on columns where values are being set	• **delete** permission on the table from which rows are being deleted *And...* • **select** permission on all columns appearing in **where** clause

Table 5-4: ANSI Permissions for update *and* delete

If you attempt to **update** or **delete** without having all the required permissions an exception is generated and the transaction is rolled back. In such a case you see the following message:

```
permission_type permission denied on object object_name database
database_name, owner owner_name
```

If this occurs, you need to be granted **select** permission on all relevant columns by the column owner.

- The **on** clause in a **grant** or **revoke** statement specifies the object on which the permission is being granted or revoked. You can grant or revoke privileges for only one table, view, or stored procedure object at a time. You can grant or revoke permissions for more than one column at a time, but all the columns must be in the same table or view.
- The keyword **public** refers to the group "public," which includes all the users of SQL Server. **public** means slightly different things for **grant** and **revoke**:
  - For **grant**, **public** includes you, the object owner. Therefore, if you have revoked permissions from yourself on your object, and later you **grant** permissions to **public**, you regain the permissions along with the rest of "public."
  - For **revoke** on object permissions, **public** excludes the owner. For **revoke** on command permissions, **public** excludes the Database Owner (who "owns" command permissions).
- The *name_list* is a list of the names of:
  - Groups
  - Users
  - A combination of users and groups, each name separated from the next by a comma
- The *role_name* is the name of a SQL Server role. This allows you to grant permissions to all users who have been granted a specific role. The roles are **sa_role** (System Administrator), **sso_role** (System Security Officer), and **oper_role** (Operator).
- The **with grant option** clause in a **grant** statement specifies that the user(s) specified in *name_list* can grant the specified object permission(s) to other users. This option can only be used for individual users, not for groups or roles. If a user has **with grant option** permission on an object, that permission is not revoked when permissions on the object are revoked from **public** or a group of which the user is a member.
- The **cascade** option in a **revoke** statement removes the specified object permissions from the user(s) specified in *name_list*, and also from any users they granted those permissions to.

You may only grant and revoke permissions on objects in the current database.

Permissions granted to roles override permissions 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 is still able to access *sales* because his role permissions override his individual permissions.

Examples: Granting and Revoking Object Permissions

This statement gives Mary and the *sales* group permission to **insert** into and **delete** from the *titles* table:

```
grant insert, delete
on titles
to mary, sales
```

This statement gives Harold permission to use the stored procedure **makelist**:

```
grant execute
on makelist
to harold
```

This statement grants permission to execute the stored procedure *sa_only* to users who have been granted the System Administrator role:

```
grant execute
on sa_only_proc
to sa_role
```

This statement gives Aubrey permission to **select**, **update**, and **delete** from the *authors* table, and to grant the same permissions to other users:

```
grant select, update, delete
on authors
to aubrey
with grant option
```

Both the following statements revoke permission for all users except the table owner to **update** the *price* and *ytd_sales* columns of the *titles* table:

```
revoke update
on titles (price, ytd_sales)
from public
```

```
revoke update(price, ytd_sales)
on titles
from public
```

This statement revokes permission for Clare to update the *authors* table, and simultaneously revokes that permission from all users she granted that permission to:

```
revoke update
on authors
from clare
cascade
```

This statement revokes permission for users who have been granted the Operator role to execute the stored procedure *new_sproc:*

```
revoke execute
on new_sproc
from oper_role
```

Examples: Granting and Revoking Command Permissions

Here are a few examples that show how to grant and revoke command permissions:

```
grant create table, create view
to mary, jane, bob
```

```
grant create table, create rule
to public
```

```
revoke create table, create rule
from mary
```

Combining *grant* and *revoke* Statements

There are two basic styles of setting up permissions in a database or on a database object. The most straightforward is to assign specific permissions to specific users.

However, if most users are going to be granted most privileges, it's easier to assign all permissions to all users and then revoke specific permissions from specific users.

For example, a Database Owner can grant all permissions on the *titles* table to all users by issuing the following statement:

```
grant all
on titles
to public
```

Then the Database Owner can issue a series of **revoke** statements, for example:

```
revoke update
on titles (royalty, advance)
from public

revoke delete
on titles
from mary, sales, john
```

Conflicting *grant* and *revoke* Statements

As implied in the previous section, **grant** and **revoke** statements are sensitive to the order in which they are issued. So, for example, if Joe's group has been granted **select** permission on the *titles* table and then Joe's permission to select the *advance* column has been revoked, Joe can select all the columns except *advance*, while the other users in his group can still select all the columns.

A **grant** or **revoke** command that applies to a group changes any conflicting permissions that have been assigned to any member of that group. For example, suppose that the owner of the *titles* table has granted different permissions to various members of the *sales* group, and then decides to standardize. He or she might issue the following statements:

```
revoke all on titles from sales
grant select on titles(title, title_id, type,
    pub_id)
to sales
```

Similarly, a **grant** or **revoke** statement issued to **public** changes, for all users, any previously issued permissions that conflict with the new regime.

The same **grant** and **revoke** statements issued in different orders can create entirely different situations. For example, the following set of statements leaves Joe without any **select** permission on *titles*:

```
grant select on titles(title_id, title) to joe
revoke select on titles from public
```

In contrast, the same statements issued in the opposite order result in only Joe having **select** permission, and only on the *title_id* and *title* columns.

Remember that when you use the keyword **public** with **grant**, you are including yourself. With **revoke** on command permissions, you are included in **public** unless you are the Database Owner. With **revoke** on object permissions, you are included in **public** unless you are the object owner. You may wish to deny yourself permission to use your own table, while giving yourself permission to access a view built on it. To do this you must issue **grant** and **revoke** statements explicitly setting your permissions. (You can always change your mind and reinstitute the permission with a **grant** statement.)

Reporting on Permissions

These system procedures provide information about command and object permissions:

- **sp_helprotect** reports on permissions on database objects or users.
- **sp_column_privileges** reports on permissions on specific columns in a table.
- **sp_table_privileges** reports on permissions on a specific table.

sp_helprotect

The system procedure **sp_helprotect** reports on permissions by database object or by user, and (optionally) by user for a specified object. Any user can execute this procedure. Its syntax is:

```
sp_helprotect name [, name_in_db [, "grant"]]
```

The first parameter, *name*, is either the name of the table, view, or stored procedure; or the name of a user, group, or role in the current database. If you specify the second parameter, *name_in_db*, only that user's permissions on the specified object are reported. If *name* is not an object, **sp_helprotect** checks whether *name* is a user, group, or role. If so, the permissions for the user, group, or role are listed. If you specify the third parameter, the keyword **grant**, and *name* is not an object, **sp_helprotect** displays all permissions granted by **with grant option.**

For example, suppose you issue the following series of **grant** and **revoke** statements:

```
grant select on titles to judy
grant update on titles to judy
revoke update on titles(contract) from judy
grant select on publishers to judy
    with grant option
```

To determine the permissions Judy now has on each column in the *titles* table, type:

```
sp_helprotect titles, judy
```

The following display results:

```
grantor grantee type   action   object     column      grantable
------- ------ -----   ------   ------     ------      -------
dbo     judy   Grant   Select   titles     All         FALSE
dbo     judy   Grant   Update   titles     advance     FALSE
dbo     judy   Grant   Update   titles     notes       FALSE
dbo     judy   Grant   Update   titles     pub_id      FALSE
dbo     judy   Grant   Update   titles     pubdate     FALSE
dbo     judy   Grant   Update   titles     title       FALSE
dbo     judy   Grant   Update   titles     title_id    FALSE
dbo     judy   Grant   Update   titles     total_sales FALSE
dbo     judy   Grant   Update   titles     type        FALSE
```

The first row of the display shows that the Database Owner ("dbo") gave Judy permission to select all columns of the *titles* table. The rest of the lines indicate that she can update only the columns listed in the display. Judy's permissions are not grantable: she cannot give **select** or **update** permissions to any other user.

To see the Judy's permissions on the *publishers* table, type:

```
sp_helprotect publishers, judy
```

In the display below, the *grantable* column indicates TRUE, meaning that Judy can grant the permission to other users.

```
grantor grantee type   action   object     column      grantable
------- ------ -----   ------   ------     ------      -------
dbo     judy   Grant   Select   publishers all         TRUE
```

sp_column_privileges

sp_column_privileges is a catalog stored procedure that returns information about permissions on columns in a table. The syntax is:

```
sp_column_privileges table_name [, table_owner
   [, table_qualifier [, column_name]]]
```

table_name is the name of the table. This is required.

table_owner can be used to specify the name of the table owner, if it is not "dbo" or the user executing **sp_column_privileges**.

table_qualifier is the name of the current database.

column_name is the name of the column on which you want to see permissions information.

Use **null** for parameters that you do not wish to specify.

For example, this statement:

```
sp_column_privileges publishers, null, null, pub_id
```

returns information about the *pub_id* column of the *publishers* table. See Volume 2, Chapter 2 of the *SQL Server Reference Manual* for more specific information about the output of **sp_column_privileges**.

sp_table_privileges

sp_table_privileges is a catalog stored procedure that returns permissions information about a specified table. The syntax is:

```
sp_table_privileges table_name [, table_owner
   [, table_qualifier]]
```

table_name is the name of the table. It is required.

table_owner can be used to specify the name of the table owner, if it is not "dbo" or the user executing **sp_column_privileges**.

table_qualifier is the name of the current database.

You can use **null** for parameters that you do not wish to specify.

For example, this statement:

```
sp_table_privileges titles
```

returns information about all permissions granted on the *titles* table. See Volume 2, Chapter 2 of the *SQL Server Reference Manual* for more specific information about the output of **sp_table_privileges**.

Permissions on Views and Stored Procedures

Views and stored procedures can serve as security mechanisms. A user can be granted permission on a view or on a stored procedure even if he or she has no permissions on objects the view or procedure references.

Views as Security Mechanisms

Through a view, users can query and modify only the data they can see. The rest of the database is neither visible nor accessible.

Permission to access the subset of data in a view must be explicitly granted or revoked, regardless of the set of permissions in force on the view's underlying tables. Data in an underlying table that is not included in the view is hidden from users who are authorized to access the view but not the underlying table.

By defining different views and selectively granting permissions on them, a user (or any combination of users) can be restricted to different subsets of data. The following examples illustrate the use of views for security purposes:

- Access can be restricted to a subset of the rows of a base table (a value-dependent subset). For example, you might define a view that contains only the rows for business and psychology books, in order to keep information about other types of books hidden from some users.
- Access can be restricted to a subset of the columns of a base table (a value-independent subset). For example, you might define a view that contains all the rows of the *titles* table, but omits the *royalty* and *advance* columns, since this information is sensitive.
- Access can be restricted to a row-and-column subset of a base table.
- Access can be restricted to the rows that qualify for a join of more than one base table. For example, you might define a view that joins the *titles*, *authors*, and *titleauthor* tables in order to display the names of the authors and the books they have written. This view would hide personal data about authors and financial information about the books.
- Access can be restricted to a statistical summary of data in a base table. For example, you might define a view that contains only the average price of each type of book.
- Access can be restricted to a subset of another view, or of some combination of views and base tables.

As an example, you want to prevent some users from accessing the columns in the *titles* table that have to do with money and sales. You could create a view of the *titles* table that omits those columns, and then give all users permission on the view but only the Sales Department permission on the table. Here's how:

```
grant all on bookview to public

grant all on titles to sales
```

An equivalent way of setting up these privilege conditions, without using a view, is this series of statements:

```
grant all on titles to public

revoke select, update on titles (price, advance,
    royalty, ytd_sales)
from public
```

```
grant select, update on titles (price, advance,
    royalty, ytd_sales)
to sales
```

One possible problem with the second scheme is that users not in the *sales* group who enter the command:

```
select * from titles
```

might be surprised to see the message that includes the phrase:

```
Permission denied
```

SQL Server expands the asterisk into a list of all the columns in the *titles* table, and since permission on some of these columns has been revoked from non-sales users, refuses access to them. The error message lists the columns for which the user does not have access.

In order to see all the columns for which they do have permission, the non-sales users would have to name them explicitly. For this reason, creating a view and granting the appropriate permissions on it is a better solution.

In addition to protecting data based on a selection of rows and / or columns, views can be used for context-sensitive protection. For example, you can create a view that gives a data entry clerk permission to access only those rows that he or she has added or updated.

In order to do so, you would add a column to a table in which the user ID of the user entering each row is automatically recorded with a default. You can define this default in the **create table** statement, like this:

```
create table testtable
    (empid      int,
    startdate   datetime,
    username    varchar(30) default user)
```

Next, define a view that includes all the rows of the table where *uid* is the current user:

```
create view context_view
as
    select *
    from testtable
    where username = user_name()
with check option
```

The rows retrievable through this view depend on the identity of the person who issues the **select** command against the view. By adding the **with check option** to the view definition, you make it impossible for any data entry clerk to falsify the information in the *username* column.

Stored Procedures as Security Mechanisms

A user with permission to execute a stored procedure can do so even if he or she does not have permissions on tables or views referenced in it. For example, a user might be given permission to execute a stored procedure that updates a row-and-column subset of a specified table, even though that user does not have any other permissions on that table.

Permissions on views and stored procedures are checked when the object is used, not when it is created.

Roles and Stored Procedures

You can use the **grant execute** command to grant execute permission on a stored procedure to all users who have been granted a specified role. Similarly, **revoke execute** removes this permission.

For further security, you can restrict the use of a stored procedure with any SQL commands you can program into it. For example, you can use the **proc_role** system function within a stored procedure to guarantee that a procedure can only be executed by users with a given role. **proc_role** returns 1 if the user has a specific role (**sa_role**, **sso_role**, or **oper_role**) and returns 0 if the user does not have that role. For example, here is a procedure that uses **proc_role** to see if the user has the System Administrator role:

```
create proc test_proc
as
if (proc_role("sa_role") = 0)
begin
    print "You don’t have the right role"
    return -1
end
else
    print "You have SA role"
    return 0
```

See “System Functions” in Volume 1 of the *SQL Server Reference Manual* for more information about **proc_role**.

Ownership Chains

Views can depend on other views and / or tables. Procedures can depend on other procedures, views, and / or tables. These dependencies can be thought of as an **ownership chain**.

Ownership Chains and Views

Typically, the owner of a view also owns its underlying objects (other views and tables), and the owner of a stored procedure owns all the procedures, tables, and views the procedure references.

Also, a view and its underlying objects are usually all in the same database, as are a stored procedure and all the objects it references. However, it is not required that all of the underlying objects be in the same database. If these objects are in different databases, a user wishing to use the view or stored procedure must be a valid or guest user in all of the databases containing the objects. (This prevents users from accessing a database unless the Database Owner has authorized it.)

When a user with permission to access a view does so, SQL Server does not check permissions on any of the view's underlying objects if:

- These objects and the view are all owned by the same user, and
- The user accessing the view is a valid user or guest in each of the databases containing the underlying objects.

Similarly, if the same user owns a stored procedure and all the views or tables it references, SQL Server checks only the permissions on the procedure.

However, if the ownership chain of a procedure or view is broken—that is, if not all the objects in the chain are owned by the same user—SQL Server checks permissions on each object in the chain whose next lower link is owned by a different user. In this way, SQL Server allows the owner of the original data to retain control over who is authorized to access it.

Ordinarily, a user who creates a view need worry only about granting permissions on that view. For example, say Mary has created a view called *auview1* on the *authors* table, which she also owns. If Mary grants **select** permission to Sue on *auview1*, SQL Server will let Sue access it without checking permissions on *authors*.

However, a user who creates a view or stored procedure that depends on an object owned by another user must be aware that any permissions he or she grants depend on the permissions allowed by those other owners.

Say Joe creates a view called *auview2*, which depends on Mary's view *auview1*. Joe grants Sue **select** permission on *auview2*.

Sue's permission	Objects	Ownership	Checks
select	auview2	Joe	Sue not owner Check permissions
select	auview1	Mary	Different owner Check permissions
none	authors	Mary	Same owner No permission check

Figure 5-1: Ownership Chains and Permission Checking for Views, Case 1

SQL Server checks the permissions on *auview2* and *auview1*, and finds that Sue can use them. SQL Server checks ownership on *auview1* and *authors* and finds they have the same owner. Sue can therefore use *auview2*.

SQL Server performs no authorization checks at the time a view is created. In fact, if Joe has permission on the **create view** command, he can define a view based on the *authors* table even if he does not have **select** permission on *authors*. However, the view would be useless to everyone, including Joe. Anyone who tried to use the view would receive an error message.

Let's take our example a step further. Suppose that Joe's view, *auview2*, depends on *auview1*, which depends on *authors*. Mary decides she likes Joe's *auview2* and creates *auview3* on top of it. Both *auview1* and *authors* are owned by Mary.

The ownership chain looks like this:

Sue's permission	Objects	Ownership	Checks
select	auview3	Mary	Sue not owner Check permissions
select	auview2	Joe	Different owner Check permissions
select	auview1	Mary	Different owner Check permissions
none	authors	Mary	Same owner No permission check

Figure 5-2: Ownership Chains and Permission Checking for Views, Case 2

When Sue tries to access *auview3*, SQL Server checks permissions on *auview3*, *auview2*, and *auview1*. If Joe has granted permission to Sue on *auview2* and Mary has granted her permission on *auview3* and *auview1*, SQL Server allows the access. SQL Server checks permissions only if the object immediately before it in the chain has a different owner (or if it is the first object in the chain). For example, it checks *auview2* because the object before it—*auview3*—is owned by a different user. It does not check permission on *authors*, because the object which immediately depends on it, *auview1*, is owned by the same user.

Procedures follow the same rules. As an example, suppose the situation is this:

Sue's permission	Objects	Ownership	Checks
execute	proc4	Mary	Sue not owner Check permissions
none	proc3	Mary	Same owner No permission check
execute	proc2	Joe	Different owner Check permissions
execute	proc1	Mary	Different owner Check permissions
none	authors	Mary	Same owner No permission check

Figure 5-3: Ownership Chains and Permission Checking for Stored Procedures

To execute *proc4*, Sue must have permission to execute *proc4*, *proc2*, and *proc1*. Permission to execute *proc3* is not necessary, since *proc3* and *proc4* have the same owner.

Ownership Chains and Stored Procedures

SQL Server checks Sue's permissions on *proc4* and all objects it references each time she executes *proc4*. SQL Server knows which referenced objects to check: it determined this the first time Sue executed *proc4*, and it saved the information with the procedure's execution plan. Unless one of the objects referenced by the procedure is dropped or otherwise redefined, SQL Server does not change its initial decision about which objects to check.

The purpose of this protection hierarchy is to allow every object's owner to fully control access to the object. Owners can control access to views and stored procedures, as well as to tables.

Triggers

Triggers are a special kind of stored procedure used to enforce integrity, especially referential integrity. (Please see the *Transact-SQL User's Guide* or the *SQL Server Reference Manual* for details.) Triggers are never executed directly, but only as a side effect of modifying a table. There is no way to **grant** or **revoke** permissions for triggers.

Only the owner of an object can create a trigger on it. However, the ownership chain can be broken if a trigger on a table references objects owned by different users. The protection hierarchy rules that apply to procedures also hold for triggers.

While the objects which a trigger affects are usually owned by the same user who owns the trigger, you can write a trigger that modifies an object owned by another user. If this is the case, any users modifying your object in a way that activates the trigger must have permission on the other object as well.

If SQL Server denies permission on a data modification command because of a trigger that affects an object for which the user does not have permission, the entire data modification command is aborted.

6 Checking Database Consistency

Introduction

The Database Consistency Checker (**dbcc**) is a set of utility commands for checking the logical and physical consistency of a database. Use the **dbcc** commands:

- As part of regular database maintenance (periodic checks run by the System Administrator). These checks can detect, and often correct, errors before they affect a user's ability to use SQL Server.
- To determine the extent of possible damage after a system error has occurred.
- Before backing up a database.
- Because you suspect that a database is damaged. For example, if using a particular table generates the message "Table corrupt", you can use **dbcc** to determine if other tables in the database are also damaged.

The integrity of the internal structures of a database depends upon the System Administrator or Database Owner running database consistency checks on a regular basis. Two major functions of **dbcc** are:

- Checking allocation structures (the commands **checkalloc**, **tablealloc**, and **indexalloc**).
- Checking page linkage and data pointers at both the page level and row level (**checktable** and **checkdb**). The next section explains page and object allocation and page linkage.

This chapter discusses:

- Page and object allocation. Understanding how SQL Server handles page and object allocation makes it easier to understand what the **dbcc** commands do.
- The **dbcc** commands: their syntax and what they do.
- When and how to use the **dbcc** commands.

All **dbcc** commands except the **dbrepair** command and **checkdb** with the **fix** option can be run while the database is active.

Errors Generated by Database Consistency Problems

Errors generated by database consistency problems usually have error numbers from 2500 through 2599 or 7900 through 7999. These messages and others that can result from database consistency problems (such as message 605) can be very scary, including phrases like "Table Corrupt" or "Extent not within segment". Many of these messages indicate severe database consistency problems, while other problems are not urgent. Some will require help from Technical Support, but many can be solved by:

- Running **dbcc** commands which have the **fix** option.
- Following instructions in the *SYBASE Troubleshooting Guide*. This guide contains step-by-step instructions for resolving many **dbcc** errors.

Whatever the techniques required to solve the problems, the solutions are much easier when you find the problem early, soon after the corruption or inconsistency originated. Consistency problems can exist on data pages that are not used frequently, such as a table that is only updated monthly. **dbcc** can find—and often fix—these problems for you before a user needs the data. The only way to find these problems early is to run **dbcc** commands often.

If you dump a database that contains consistency errors, and load that dump of the database at a later time, the result will be a database that has consistency problems, since a database dump is a logical image of the data pages.

dbcc Permissions

The Database Owner and System Administrator are automatically granted permission to use all of the **dbcc** commands. Permission for **dbcc checktable**, **dbcc reindex**, and **dbcc fix_text** defaults to the table owner. Permission to run **dbcc** is not transferable.

Page and Object Allocation Concepts

When you initialize a database device, the **disk init** command divides the new space into **allocation units** of 256 2K data pages (data pages on Stratus machines are 4K each). The first page of each allocation unit is an **allocation page**, which tracks the use of all pages in the allocation unit. Allocation pages have an object ID of 99; they aren't real database objects, and don't appear in system tables, but **dbcc** errors on allocation pages report this value.

Whenever a table or index requires space, SQL Server allocates a block of 8 2K pages (8 4K pages on Stratus) to the object. This 8-page block is called an **extent**. Each 256-page allocation unit contains 32 extents. SQL Server use extents as a unit of space management to allocate and de-allocate space:

- When you create a table or index SQL Server allocates an extent for the object.
- When you add rows to an existing table, SQL Server allocates another page if existing pages are full. If all of the pages in an extent are full, SQL Server allocates an additional extent.
- When you drop a table or index, SQL Server deallocates the extents it occupied.
- When you delete rows from a table, so that it shrinks off a page, SQL Server deallocates the page; if the table shrinks off the extent, SQL Server deallocates the extent.

Every time space is allocated or deallocated on an extent, SQL Server records the event on the allocation page that tracks the extents for that object. This provides a fast method for tracking space allocations in the database, since objects can shrink or grow without a lot of overhead.

Table 6-1 shows how data pages are set up within extents and allocation units in SQL Server databases.

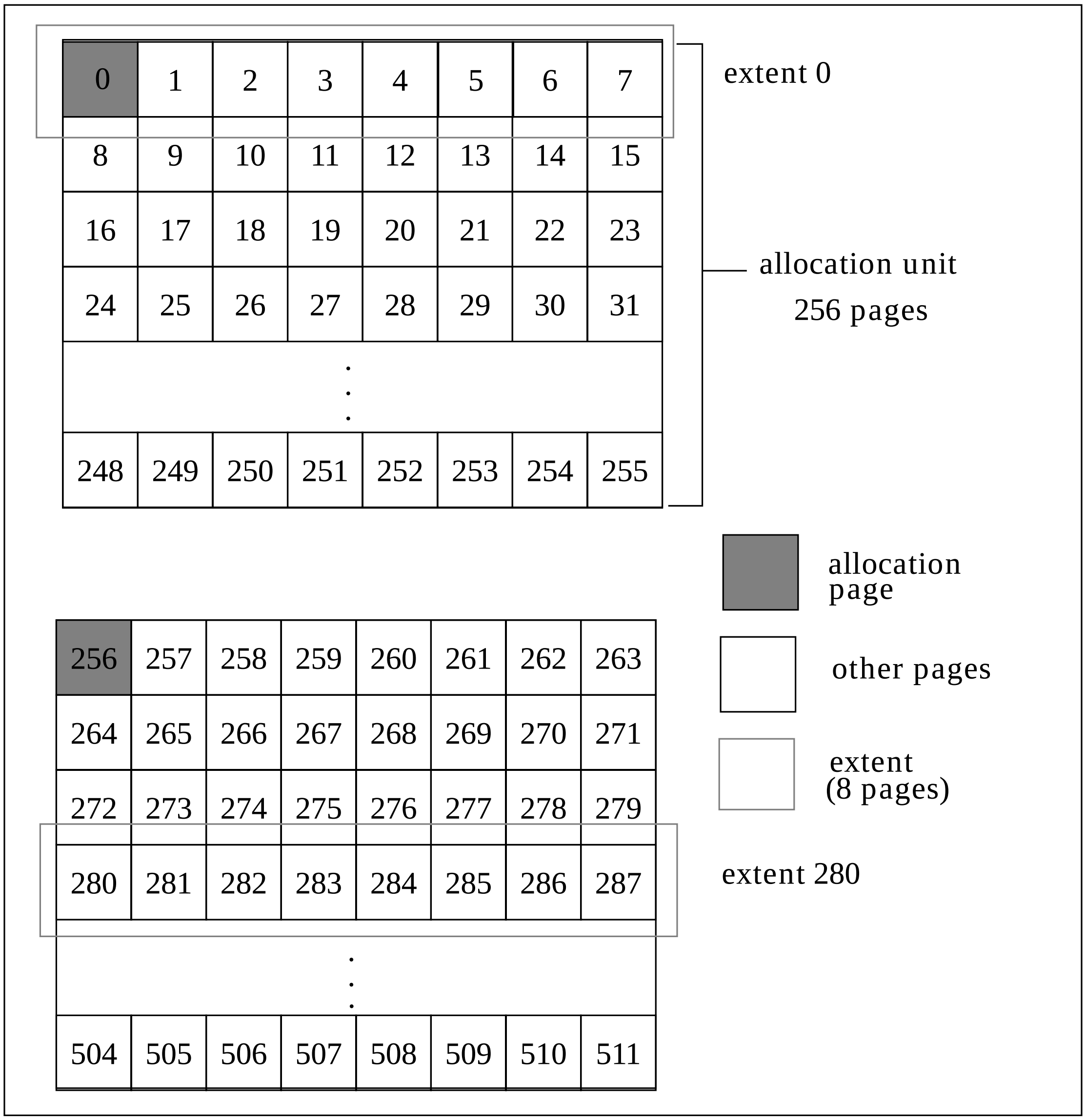


Figure 6-1: Page Management with Extents

dbcc provides the **checkalloc** command for checking all allocation pages in the database, and the **indexalloc** and **tablealloc** commands for checking allocation for specific database objects.

dbcc checkalloc checks all allocation pages (page 0 and all pages divisible by 256) in a database, and reports on the allocation information it finds there.

The Object Allocation Map (OAM)

Each table, and each index on a table, has an **Object Allocation Map** (**OAM**). The OAM is stored on pages allocated to the table or index and is checked when a new page is needed for the index or table. A single OAM page can hold allocation mapping for between 2016 and 64,260 data or index pages.

The OAM pages point to the allocation page for each allocation unit where the object uses space. The allocation pages, in turn, track the information about extent and page usage within the allocation unit. In other words, if the *titles* table is stored on extents 24 and 272, the OAM page for the *titles* table points to pages 0 and 256.

Figure 6-2: OAM Page and Allocation Page Pointers shows an object stored on 4 extents, numbered 0, 24, 272 and 504. The Object Allocation Map is stored on the first page of the first segment. In this case, since the allocation page occupies page 0, the OAM is located on page 1.

This OAM points to two allocation pages: page 0 and page 256.

These allocation pages track the pages used in each extent used by all objects with storage space in the allocation unit. For the object in this example, it tracks the allocation and deallocation of pages on extents 0, 24, 272, and 504.

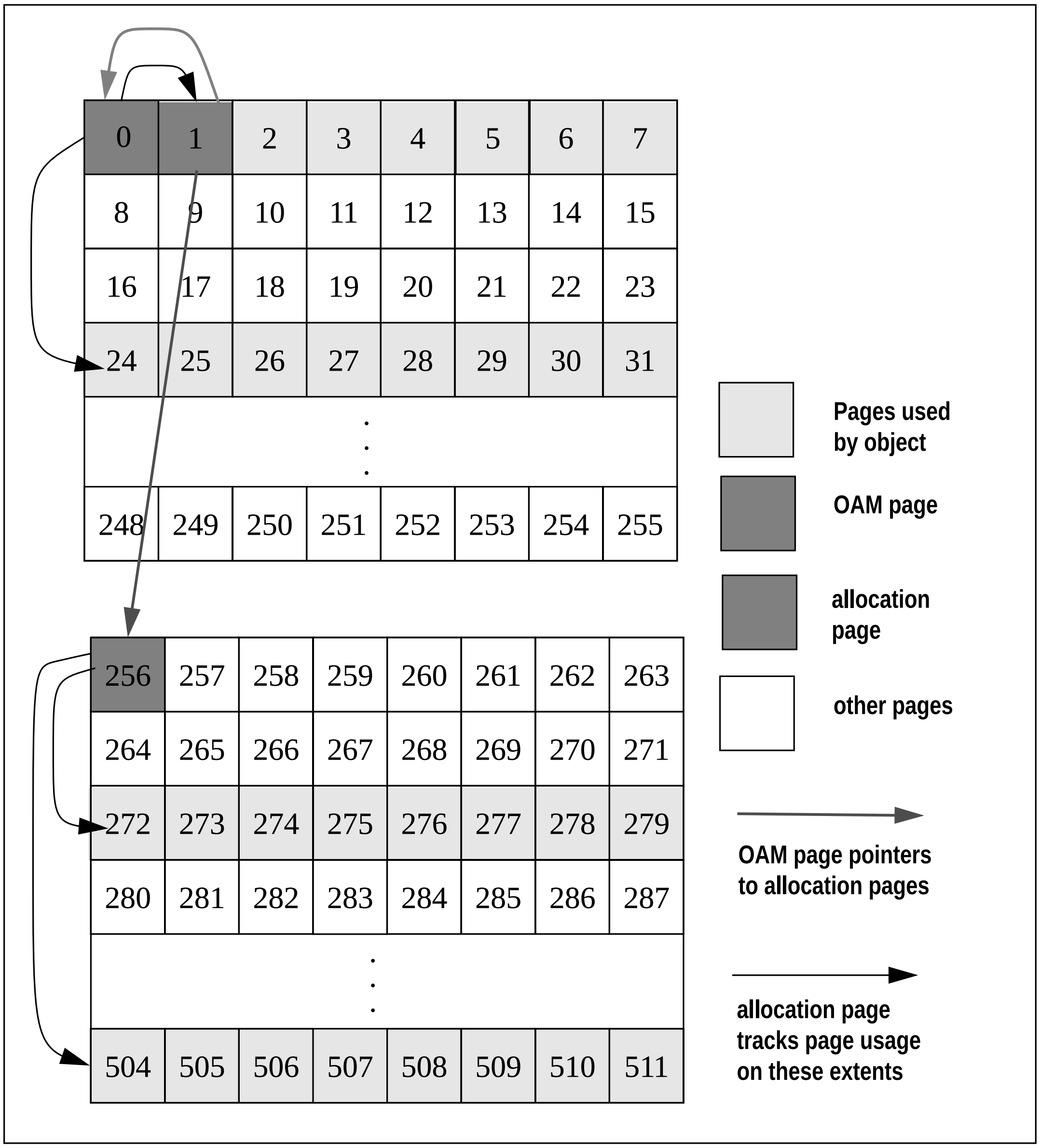


Figure 6-2: OAM Page and Allocation Page Pointers

The **dbcc checkalloc** and **dbcc tablealloc** commands examine this OAM page information, in addition to checking page linkage, described in the following section.

Page Linkage

Once a page has been allocated to a table or index, that page is linked with other pages used for the same object. *Figure 6-3: How a Newly-Allocated Page is Linked with Other Pages* illustrates this linking. Each page contains a header that includes the number of the page that precedes it (prev) and of the page that follows it (next). When a new page is allocated, the header information on the surrounding pages changes to point to that page. **dbcc checktable** and **dbcc checkdb** check page linkage. **dbcc checkalloc**, **tablealloc**, and **indexalloc** compare page linkage to information on the allocation page.

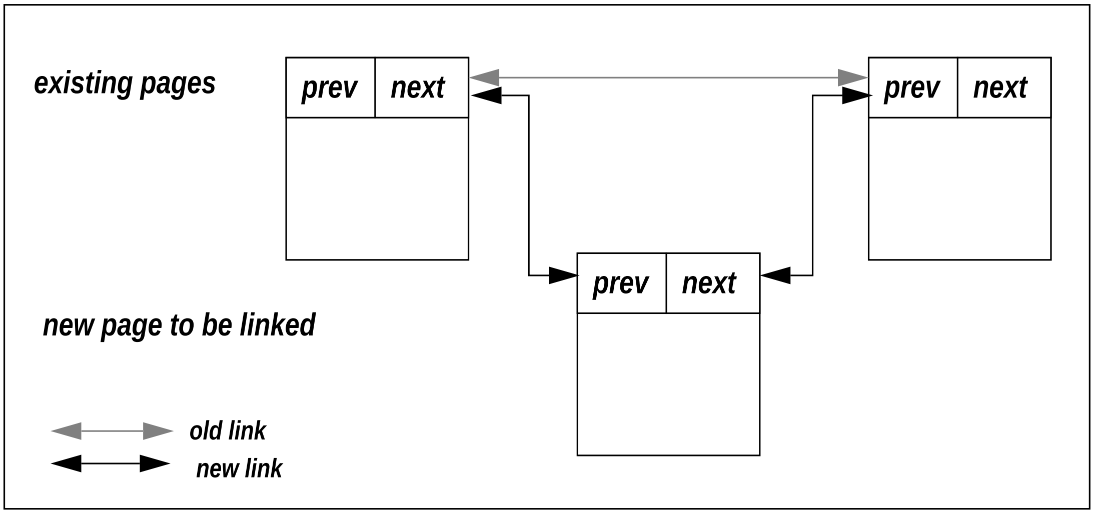


Figure 6-3: How a Newly-Allocated Page is Linked with Other Pages

Individual *dbcc* Commands

dbcc checktable

The syntax for **dbcc checktable** is:

```
dbcc checktable ({table_name | table_id}
   [, skip_ncindex] )
```

The **dbcc checktable** command checks the specified table to see that:

- Index and data pages are correctly linked
- Indexes are in properly sorted order
- All pointers are consistent

- The data rows on each page all have entries in the first OAM page matching their respective locations on the page

The option **skip_ncindex** allows you to skip checking the page linkage, pointers and sort order on nonclustered indexes. The linkage and pointers of clustered indexes and data pages is essential to the integrity of your tables. You can easily drop and recreate nonclustered indexes if SQL Server reports problems with page linkage or pointers.

dbcc checktable can be used with the table name or the table's object ID. The *sysobjects* table stores this information in the *name* and *id* columns.

Here's an example of a report on an undamaged table:

```
1> dbcc checktable(titles)
2> go
```

```
Checking titles
The total number of data pages in this table is 2.
Table has 18 data rows.

DBCC execution completed. If DBCC printed error
messages, see your System Administrator.
```

To check a table not in the current database, supply the database name. To check a table owned by another object, supply the owner's name. You must enclose any qualified table name in quotes:

```
dbcc checktable("pubs2.newuser.testtable")
```

These are the problems that **dbcc checktable** addresses:

- If the page linkage is incorrect, **dbcc checktable** displays an error message.
- If the sort order (*sysindexes.soid*) or character set (*sysindexes.csid*) for a table with columns with *char* or *varchar* datatypes is incorrect, and the table's sort order is compatible with SQL Server's default sort order, **dbcc checktable** corrects the values for the table. Only the binary sort order is compatible across character sets.

➤ ***Note***

If you change sort orders, character-based user indexes are marked read-only and must be checked and rebuilt if necessary. See Chapter 17, "Language, Sort Order, and Character Set Issues", for more information about changing sort orders.

- If there are data rows not accounted for in the first OAM page for the object, **dbcc checktable** updates the number of rows on that page. This is never a serious problem, it is just a little quick housekeeping. The built-in function **rowcnt** uses this value to provide fast row estimates in procedures like **sp_spaceused**.

dbcc checkdb

The syntax for **dbcc checkdb** is:

```
dbcc checkdb [(database_name [, skip_ncindex]) ]
```

The **dbcc checkdb** command runs the same checks as **dbcc checktable** on each table in the specified database. If you do not give a database name, **dbcc checkdb** checks the current database. **dbcc checkdb** gives similar messages to those returned by **dbcc checktable** and makes the same types of corrections.

If you specify the optional **skip_ncindex** mode, **dbcc checkdb** does not check any of the nonclustered indexes on user tables in the database.

dbcc checkcatalog

The syntax for **dbcc checkcatalog** is:

```
dbcc checkcatalog [(database_name)]
```

The **dbcc checkcatalog** command checks for consistency within and between the system tables found in a particular database. If no database name is given, **dbcc checkcatalog** checks the current database.

For example, it verifies that:

- Every type in *syscolumns* has a matching entry in *systypes*
- Every table and view in *sysobjects* has at least one column in *syscolumns*
- The last checkpoint in *syslogs* is valid

It also lists the segments defined for use by the database.

Here’s an example of output from **dbcc checkcatalog**:

```
dbcc checkcatalog (testdb)
```

```
Checking testdb
The following segments have been defined for database 5
(database name testdb).
virtual start addr      size      segments
--------------------    ------    --------------------------
33554432                4096
                                   0
                                   1
16777216                1024
                                   2
DBCC execution completed. If DBCC printed error messages, see
your System Administrator.
```

dbcc checkalloc

The syntax for **dbcc checkalloc** is:

dbcc checkalloc [(*database_name* [, fix | nofix])]

The **dbcc checkalloc** command checks the specified database to see that:

- All pages are correctly allocated
- No page is allocated that is not used
- No page is used that is not allocated

If you do not give a database name, **dbcc checkalloc** checks the current database.

The default mode for **dbcc checkalloc** is **nofix**. To use **dbcc checkalloc** with the **fix** option, you must first put the database into single-user mode with the command:

sp_dboption dbname, "single user", true

You can only issue this command when no one is using the database.

dbcc checkalloc reports the amount of space allocated and used. **dbcc checkalloc** output consists of a block of data for each table, including the system tables and the indexes on each table. For each table or index, it reports the number of pages and extents used. Table information is reported as either INDID =0 or INDID=1. Tables without clustered indexes have *indid* = 0 (for example, *salesdetail*, below). Tables with clustered indexes have *indid* = 1, and the report for these includes information on the data level and the index level. See the report for *titleauthor* and *titles*, below. Nonclustered indexes are numbered consecutively, starting from INDID=2.

Here is a portion of a report on *pubs2*, showing the output for three tables:

```
***************************************************************
TABLE: salesdetail              OBJID = 144003544
INDID=0  FIRST=297       ROOT=299        SORT=0
        Data level: 0.  3 Data  Pages in 1 extents.
INDID=2  FIRST=289       ROOT=290        SORT=1
        Indid     : 2.  3 Index Pages in 1 extents.
INDID=3  FIRST=465       ROOT=466        SORT=1
        Indid     : 3.  3 Index Pages in 1 extents.
TOTAL # of extents = 3
***************************************************************
TABLE: titleauthor              OBJID = 176003658
INDID=1  FIRST=433       ROOT=425        SORT=1
        Data level: 1.  1 Data  Pages in 1 extents.
        Indid     : 1.  1 Index Pages in 1 extents.
INDID=2  FIRST=305       ROOT=305        SORT=1
        Indid     : 2.  1 Index Pages in 1 extents.
INDID=3  FIRST=441       ROOT=441        SORT=1
        Indid     : 3.  1 Index Pages in 1 extents.
TOTAL # of extents = 4
***************************************************************
TABLE: titles           OBJID = 208003772
INDID=1  FIRST=417       ROOT=409        SORT=1
        Data level: 1.  3 Data  Pages in 1 extents.
        Indid     : 1.  1 Index Pages in 1 extents.
INDID=2  FIRST=313       ROOT=313        SORT=1
        Indid     : 2.  1 Index Pages in 1 extents.
TOTAL # of extents = 3
***************************************************************
```

In its default (**nofix**) mode, **dbcc checkalloc** does not correct allocation errors. In **fix** mode, it can fix all of the allocation errors fixed by **dbcc tablealloc**, and can also fix pages which remain allocated to objects that have been dropped from the database.

Since the database must be in single-user mode to use the **fix** option, you can run **dbcc checkalloc** in nofix mode, and use **dbcc tablealloc** or **dbcc indexalloc**, described below, with the **fix** option to correct errors found in individual tables or indexes.

dbcc tablealloc

The syntax for **dbcc tablealloc** is:

```
dbcc tablealloc ({table_name | table_id}
   [, {full | optimized | fast | null}
   [, fix | nofix]])
```

➤ ***Note***

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

This command performs the same checks as **dbcc checkalloc** on a single table. You can specify either the table name or the table's object ID (the *id* column from *sysobjects).*

There are three types of reports that can be generated with **dbcc tablealloc**: **full, optimized,** and **fast.**

- The **full** option is equivalent to **dbcc checkalloc** at a table level; it reports all types of allocation errors.
- The **optimized** option 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 not listed in the OAM pages. If the type of report is not indicated in the command, or if you indicate **null, optimized** is the default.
- The **fast** option produces an exception report of pages that are referenced but not allocated in the extent (2521 errors). It does not produce an allocation report.

Table 6-1: dbcc Commands Compared for Speed, Thoroughness, and Locking on page 6-15 compares these options and other **dbcc** commands for speed, completeness, locking, and performance issues.

Correcting Allocation Errors—the fix | nofix option

The **fix** | **nofix** option determines whether or not **tablealloc** fixes the allocation errors found in the table. The default is **fix** for all user tables; the default for all system tables is **nofix.** To use the **fix** option with system tables, you must first put the database into single user mode.

➤ ***Note***

The number of errors corrected when you run **dbcc tablealloc** and **dbcc indexalloc** with the **fix** option depends on the type of report (**full**, **optimized**, or **fast**) you request. Running **dbcc tablealloc** with the **full** option fixes more errors than running the command with the **optimized** or **fast** option.

Output from a **dbcc tablealloc** command with the **fix** option displays any allocation errors found, as well as any corrections made. Here is an example of an error message that appears whether or not the **fix** option is used:

```
Msg 7939, Level 22, State 1:
Line 2:
Table Corrupt: The entry is missing from the OAM
for object id 144003544 indid 0 for allocation
page 2560.
```

The following message, which appears when the **fix** option is used, indicates that the missing entry has been restored:

```
The missing OAM entry has been inserted.
```

dbcc indexalloc

This is the syntax for **dbcc indexalloc**:

dbcc indexalloc ({*table_name* | *table_id* }, *index_id*
[, {full | optimized | fast | null}
[, fix | nofix]])

➤ ***Note***

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

The **dbcc indexalloc** command checks the specified index to see that:

- All pages are correctly allocated
- No page is allocated that is not used
- No page is used that is not allocated

This is an index-level version of **dbcc checkalloc**, providing the same integrity checks on an individual index. You can specify either the table name or the table's object ID (the *id* column from *sysobjects)* and the index's *indid* from *sysindexes*. **dbcc checkalloc** and **dbcc indexalloc** include the index ID's in their output.

dbcc indexalloc produces the same three types of reports as **dbcc tablealloc**: **full**, **optimized**, and **fast** (see preceding paragraphs on **tablealloc**). The **fix|nofix** option works the same in **dbcc indexalloc** as in **dbcc tablealloc.**

dbcc dbrepair

This is the syntax for **dbcc dbrepair**:

dbcc dbrepair (*database_name*, dropdb)

The **dbcc dbrepair dropdb** command drops a damaged database. The Transact-SQL command **drop database** does not work on a database that cannot be recovered or used. Issue the **dbrepair** statement from the *master* database; no users, including the **dbrepair** user, can be using the database when it is dropped.

dbcc reindex

The System Administrator or table owner should run **dbcc reindex** after changing SQL Server's sort order. This is the syntax for **dbcc reindex**:

dbcc reindex ({*table_name* | *table_id*})

The **dbcc reindex** command checks the integrity of indexes on user tables by running a "fast" version of **dbcc checktable**. It drops and rebuilds indexes it suspects are corrupted (that is, the order in which the indexes sort data is not consistent with the new sort order). For more information, see Chapter 17, ''Language, Sort Order, and Character Set Issues''.

dbcc fix_text

The **dbcc fix_text** command upgrades *text* values after a SQL Server's character set has been changed to a multibyte character set. This is the syntax for **dbcc fix_text**:

dbcc fix_text ({*table_name* | *table_id*})

Changing to a multibyte character set makes the management of *text* data more complicated. Since a *text* value can be large enough to cover several pages, SQL Server must be able to handle characters that may span page boundaries. To do so, SQL 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 Chapter 17, ''Language, Sort Order, and Character Set Issues'', for more information.

How, When, and Why to Use the *dbcc* Commands

The following sections compare the **dbcc** commands, provide suggestions for scheduling and strategies to avoid serious performance impacts, and provide additional information about **dbcc** output.

Comparing the *dbcc* Commands

Run the following **dbcc** commands whenever you schedule database maintenance:

- **dbcc checkdb** or **dbcc checktable**
- **dbcc checkalloc** or **dbcc indexalloc** and **dbcc tablealloc**
- **dbcc checkcatalog**

In general, the more thoroughly the **dbcc** command checks the integrity of an object or database, the slower it is. Table 6-1 compares the speed, thoroughness, level of checking, and locking and other performance implications of these commands. Bear in mind that **dbcc checktable** (and **dbcc checkdb**) and **dbcc checkcatalog** perform different types of integrity checks than **dbcc checkalloc**, **dbcc tablealloc**, and **dbcc indexalloc.**

Command and Option	Level	Locking and Performance	Speed	Thoroughness
checktable **checkdb**	page chains, sort order, data rows for all indexes	shared table lock; **dbcc checkdb** locks one table at a time and releases lock after it finishes checking that table	slow	high
checktable **checkdb** with **skip_ncindex**	page chains, sort order, data rows for tables and clustered indexes	same as above	up to 40% faster than without the **skip_ncindex** option	medium
checkalloc	page chains	no locking; performs a lot of I/O and may saturate the I/O calls; only allocation pages are cached	slow	high

Table 6-1: dbcc Commands Compared for Speed, Thoroughness, and Locking

Command and Option	Level	Locking and Performance	Speed	Thoroughness
tablealloc full **indexalloc full**	page chains	shared table lock; performs a lot of I/O; only allocation pages are cached	slow	high
tablealloc optimized **indexalloc optimized**	allocation pages	shared table lock; perform a lot of I/O; only allocation pages are cached	moderate	medium
tablealloc fast **indexalloc fast**	OAM pages	shared table lock	fastest	low
checkcatalog	rows in system tables	shared page locks on system catalogs; releases lock after each page is checked; very few pages cached		

Table 6-1: dbcc Commands Compared for Speed, Thoroughness, and Locking (continued)

Scheduling Database Maintenance at Your Site

Several factors determine how often you should run **dbcc** commands in databases at your site and which ones you need to run:

- When are your databases heavily used? Is your installation used heavily primarily between the hours of 8 a.m. and 5 p.m., or is it used heavily 24 hours a day?

  If your SQL Server is used primarily between the hours of 8 a.m. and 5 p.m., Monday through Friday, you can run **dbcc** checks at night and on weekends so the checks do not have a significant impact on your users. If your tables are not extremely large, you can run a complete set of **dbcc** commands fairly frequently.

  If your SQL Server is heavily used 24 hours a day, 7 days a week, you may want to schedule a cycle of checks on individual tables and indexes using **dbcc checktable**, **dbcc tablealloc**, and **dbcc indexalloc**. At the end of the cycle, when all tables have been checked, you can run **dbcc checkcatalog** and back up the database.

  Some sites with 24-hour, high-performance demands choose to run **dbcc** checks by:

  - Dumping the database to tape
  - Loading their database dumps into a separate SQL Server

- Running **dbcc** commands on the copied database
- Running **dbcc** commands with the **fix** on appropriate objects in the original database, if errors are detected that can be repaired with the **fix** options

Since the dump is a logical copy of the database pages, any problems in the original database are duplicated in the dumped copy.

You should run **dbcc checkdb** regularly on the entire database, or **dbcc checktable** on each table in the database. You might choose to run the object-level commands rather than the database-level check for performance reasons. **dbcc checkdb** acquires locks on the objects while it checks them, and you cannot control the order in which it checks the objects. For example, if you have one application running that uses *table4*, *table5*, and *table6*, and the **dbcc** checks take 20 minutes to complete, the application might be blocked for a long time while you are using **dbcc checkdb**.

Using the table-level commands, you can intersperse checks on tables not used in the application, thereby allowing the application access in the intervals:

```
dbcc checktable(table4)
dbcc checktable(table1)  /*not in application*/
dbcc checktable(table5)
dbcc checktable(table2)  /*not in application*/
dbcc checktable(table6)
```

- How often do you back up your databases with **dump database**?

  The more often you back up your databases and dump your transaction logs, the more data you can recover in case of failure. You and the users at your site need to decide how much data you are willing to lose in the event of a disaster and develop a dump schedule to support that decision. After you schedule your dumps, decide how to incorporate the **dbcc** commands into that schedule.

  An ideal time to dump a database is after you run a complete check of that database using **dbcc checkalloc**, **dbcc checkcatalog**, and **dbcc checkdb**. If these commands find no errors in the database, you know that your backup contains a clean database. You can use scripts to automate this process, sending output to a file and using an operating system script to search the file for errors. Use **dbcc tablealloc** or **indexalloc** on individual tables and indexes to correct allocation errors reported by **dbcc checkalloc**.

- How many tables contain highly critical data and how often does that data change? How large are those tables?

  If you have only a few tables containing critical data or data that changes often, you may want to run the table- and index-level **dbcc** commands more frequently on those tables.

What to Look for in *dbcc* Output

The output of most **dbcc** commands includes information that identifies the object or objects being checked, and error messages that indicate what problems, if any, the command finds in the object. When **dbcc tablealloc** and **dbcc indexalloc** are run with the default **fix** option, the output also indicates the repairs that the command makes. Here's an annotated example of **dbcc tablealloc** output for a table with an allocation error:

```
dbcc tablealloc(table5)
```

Information from *sysindexes* about object being checked:

```
TABLE: table5            OBJID = 144003544
INDID=0  FIRST=337       ROOT=2587       SORT=0
```

Error message:

```
Msg 7939, Level 22, State 1:
Line 2:
Table Corrupt: The entry is missing from the OAM for object id
144003544 indid 0 for allocation page 2560.
```

Message indicating that error has been corrected:

```
The missing OAM entry has been inserted.
        Data level: 0.  67 Data  Pages in 9 extents.
```

***dbcc* report on page allocation:**

```
TOTAL # of extents = 9
Alloc page 256 (# of extent=1 used pages=8 ref pages=8)
EXTID:560 (Alloc page: 512) is initialized.  Extent follows:
NEXT=0 PREV=0 OBJID=144003544 ALLOC=0xff DEALL=0x0 INDID=0 STATUS=0x0
Alloc page 512 (# of extent=2 used pages=8 ref pages=8)
Page 864 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0x1)
Page 865 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0x3)
Page 866 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0x7)
Page 867 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0xf)
Page 868 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0x1f)
Page 869 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0x3f)
Page 870 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0x7f)
Page 871 allocated (Alloc page: 768 Extent ID: 864 Alloc mask: 0xff)
Alloc page 768 (# of extent=1 used pages=8 ref pages=8)
Alloc page 1024 (# of extent=1 used pages=8 ref pages=8)
Alloc page 1280 (# of extent=1 used pages=8 ref pages=8)
Alloc page 1536 (# of extent=1 used pages=8 ref pages=8)
Alloc page 1792 (# of extent=1 used pages=8 ref pages=8)
Alloc page 2048 (# of extent=1 used pages=8 ref pages=8)
```

(Output deleted....)

Information on resources used:

```
Statistical information for this run follows:

Total # of pages read = 68
Total # of pages found cache = 68
Total # of physical reads = 0
Total # of saved I/O = 0
```

Message printed on completion of *dbcc* command:

```
DBCC execution completed. If DBCC printed error messages, see your System
Administrator.
```

7 Developing a Backup and Recovery Plan

Introduction

SQL Server has **automatic recovery** procedures to protect you from power outages and computer failures. To protect yourself against media failure, you must make regular and frequent backups of your databases.

This chapter is part of a four-chapter unit on **backup** and **recovery**. It provides information to help you develop a backup and recovery plan. The overview of SQL Server's backup and recovery processes in this chapter includes:

- How the transaction log keeps track of database changes
- How checkpoints synchronize the database and transaction log
- The function of the **dump database**, **dump transaction**, **load database**, and **load transaction** commands
- The role of the Backup Server in the backup and recovery process
- SQL Server's automatic recovery process

It also covers backup and recovery issues you should address before you begin using your system for production, such as:

- Choosing backup media
- Configuring your system for backup and recovery
- Creating logical names for local backup devices
- Scheduling backups of your system and user databases
- Granting permission to use the commands for backup and recovery

For more information about	See
dump, **load**, and **sp_volchanged** syntax	Chapter 8, "Backing Up and Restoring User Databases"
Backing up and restoring the system databases	Chapter 9, "Backing Up and Restoring the System Databases"
Using thresholds to automate backups	Chapter 10, "Managing Free Space with Thresholds"

Table 7-1: Further Information About Backup and Recovery

Keeping Track of Database Changes

SQL Server uses transactions to keep track of all database changes. Transactions are SQL Server's units of work. A transaction consists of one or more Transact-SQL statements that succeed—or fail—as a unit.

Each SQL statement that modifies data is considered a **transaction**. Users can also define transactions by enclosing a series of statements within a **begin transaction ... end transaction** block. For more information about transactions, see "Transactions" in Volume 1 of the *SQL Server Reference Manual*.

Each database has its own **transaction log**, the system table *syslogs*. The transaction log automatically records every transaction issued by each user of the database. You cannot turn off transaction logging.

The transaction log is a **write-ahead log**. When a user issues a statement that would modify the database, SQL Server automatically writes the changes to the log. After all changes for a statement have been recorded in the log, they are written to an in-cache copy of the data page. The data page remains in cache until the memory is needed for another database page. At that time, it is written to disk.

If any statement in a transaction fails to complete, SQL Server reverses all changes made by the transaction. SQL Server writes an "end transaction" record to the log at the end of each transaction, recording the status (success or failure) of the transaction.

Getting Information about the Transaction Log

The transaction log contains enough information about each transaction to ensure that it can be recovered. Use the **dump transaction** command to copy the information it contains to tape or disk. Use **sp_spaceused syslogs** to check the size of the log, or **sp_helpsegment logsegment** to check the space available for log growth.

◆ ***WARNING!***

Never use insert, update, or delete commands to modify *syslogs*.

Synchronizing a Database and its Transaction Log: Checkpoints

A checkpoint writes all “dirty” pages—pages that have been modified in memory, but not on disk, since the last checkpoint—to the database device. SQL Server’s automatic **checkpoint** mechanism guarantees that data pages changed by completed transactions are regularly written from the cache in memory to the database device. Synchronizing the database and its transaction log shortens the time it takes to recover the database after a system crash. Figure 12-1 illustrates the checkpoint process.

Setting the Recovery Interval

Typically, the automatic recovery process takes from a few seconds to a few minutes per database. Time varies depending on the size of the database, the size of the transaction log, and the number and size of transactions that must be committed or rolled back.

Use the **recovery interval** configuration variable to specify the maximum permissible recovery time, in minutes. SQL Server runs automatic checkpoints often enough to recover the database within that period of time.

Use **sp_configure** for a report on the current recovery interval:

```
sp_configure "recovery interval"
```

The default value, 5, allows recovery within 5 minutes per database. Use **sp_configure “recovery interval”** followed by the **reconfigure** command to change the recovery interval. For example, to set the recovery interval to 3 minutes, issue these commands:

```
sp_configure "recovery interval", 3

reconfigure
```

➤ ***Note***

The recovery interval has no effect on long-running transactions that are active at the time SQL Server fails. It may take as much time to reverse these transactions as it took to run them.

The Automatic Checkpoint Procedure

Approximately once each minute, the checkpoint process "wakes up" and checks each database on the server to see how many records have been added to the transaction log since the last checkpoint. If the server estimates that the time that would be required to recover these transactions is greater than the database's recovery interval, SQL Server issues a checkpoint.

The modified pages are written from cache onto the database devices, and the checkpoint event is recorded in the transaction log. Then, the checkpoint process sleeps for another minute.

To see the checkpoint process, execute **sp_who**. The checkpoint process usually displays as "CHECKPOINT SLEEP" in the *cmd* column:

```
spid  status     loginame   hostname blk dbname  cmd
----- ---------- ---------- -------- --- ------- ----------------
    1 running    sa         mars     0   master  SELECT
    2 sleeping   NULL                0   master  NETWORK HANDLER
    3 sleeping   NULL                0   master  MIRROR HANDLER
    4 sleeping   NULL                0   master  CHECKPOINT SLEEP
```

Truncating the Log After Automatic Checkpoints

System Administrators can use the **trunc log on chkpt** database option to truncate the transaction log every time SQL Server performs an automatic checkpoint. Execute this command from the *master* database:

```
sp_dboption database_name, "trunc log on chkpt",
true
```

This option is not suitable for production environments because it does not make a copy of the transaction log before truncating it. Use **trunc log on chkpt** only for:

- Databases whose transaction logs cannot be backed up because they are not on a separate segment
- Test databases for which current backups are not important

➤ ***Note***

If you leave the **trunc log on chkpt option** off (the default condition) the transaction log continues to grow until you truncate it with the **dump transaction** command.

To protect your log from running out of space, you should design your last-chance threshold procedure to dump the transaction log. For more information about threshold procedures, see Chapter 10, "Managing Free Space with Thresholds".

Manually Requesting a Checkpoint

Database Owners can issue the **checkpoint** command to force all modified pages in memory to be written to disk. Manual checkpoints do not truncate the log, even if the **trunc log on chkpt** option of **sp_dboption** is turned on.

Use the **checkpoint** command:

- As a precautionary measure in special circumstances—for example, just before a planned **shutdown with nowait** so that SQL Server's recovery mechanisms will occur within the recovery interval. (An ordinary **shutdown** performs a checkpoint.)
- To cause a change in database options to take effect after executing the **sp_dboption** system procedure. (After you run **sp_dboption**, an informational message reminds you to run **checkpoint**.)

Automatic Recovery After a System Failure or Shutdown

Each time you restart SQL Server—for example, after a power failure, an operating system failure, or the use of the **shutdown** command—it automatically performs a set of recovery procedures on each database.

The recovery mechanism compares each database to its transaction log. If the log record for a particular change is more recent than the data page, the recovery mechanism reapplies the change from the transaction log. If a transaction was ongoing at the time of the failure, the recovery mechanism reverses all changes that were made by the transaction. This mechanism ensures that the entire transaction succeeds or fails as a unit.

When SQL Server is booted, it performs database recovery in this order:

1. Recovers *master*
2. Recovers *sybsecurity*
3. Recovers *model*

4. Creates *tempdb* (by copying *model*)
5. Recovers *sybsystemprocs*
6. Recovers user databases, in order by *sysdatabases.dbid*

Users can log into SQL Server as soon as the system databases have been recovered, but cannot access other databases until they have been recovered.

Determining Whether Messages Are Displayed During Recovery

The configuration variable **recovery flags** determines whether SQL Server displays detailed messages about each transaction on the console screen during recovery. By default, these messages are not displayed. To display messages, issue these commands:

```
sp_configure "recovery flags", 1
reconfigure
```

Using the Dump and Load Commands

In case of media failure, such as a disk crash, you can restore your databases if—and only if—you have been making regular backups of the databases and their transaction logs. Full recovery depends on the regular use of the **dump database** and **dump transaction** commands to back up databases and the **load database** and **load transaction** commands to restore them. These commands are described briefly below and more fully in Chapter 8, "Backing Up and Restoring User Databases" and Chapter 9, "Backing Up and Restoring the System Databases".

◆ ***WARNING!***

Never use operating system copy commands to copy a database device. Loading the copy into SQL Server causes massive database corruption.

Checking Database Consistency: *dbcc*

The dump commands can complete successfully even if your database is corrupt. Before you back up a database, use the **dbcc** commands to check its consistency. See Chapter 6, "Checking Database Consistency", for more information.

Making Routine Database Dumps: *dump database*

The **dump database** command makes a copy of the entire database, including both the data and the transaction log. **dump database** does **not** truncate the log.

dump database allows **dynamic dumps**: users can continue to make changes to the database while the dump takes place. This feature makes it convenient to back up databases on a regular basis.

The **dump database** command executes in three phases. A progress message informs you when each phase completes. When the dump is finished, it reflects all changes that were made during its execution, except for those initiated during Phase 3.

Making Routine Transaction Log Dumps: *dump transaction*

Use the **dump transaction** command (or its abbreviation **dump tran**) to make routine backups of your transaction log. **dump transaction** is similar to the incremental backups provided by many operating systems. It copies the transaction log, providing a record of any database changes made since the last database or transaction log dump. Once **dump transactio**n has copied the log, it truncates the inactive portion.

dump transaction takes less time and storage space than a full database backup, and is usually run more often. Users can continue to make changes to the database while the dump is taking place. You can run **dump transaction** only if the database stores its log on a separate segment.

After a media failure, use the **with no_truncate** option of **dump transaction** to backup your transaction log. This provides a record of the transaction log up to the time of the failure.

Copying the Log After Device Failure: *dump tran with no_truncate*

When your data device fails and the database is inaccessible, use the special **with no_truncate** option of the **dump transaction** command to get a current copy of the log. This option does not truncate the log. You can use it only if the transaction log is on a separate segment and the *master* database is accessible.

Restoring the Entire Database: *load database*

Use the **load database** command to load the backup created with **dump database**. You can load the dump into a pre-existing database, or create a new database with the **for load** option. When you create a new database, allocate at least as much space as was allocated to the original database.

◆ ***WARNING!***

You cannot load a dump that was made on a different platform or with an earlier version of SQL Server.

If the database you are loading includes tables that contain the primary keys for tables in other databases, you must load the dump into the same database name.

Put the database into single-user mode so that users cannot make changes from the time you begin loading it until you have applied the last transaction log dump. Before loading a database, lock it by using **sp_dboption** to set the **no chkpt on recovery, dbo use only,** and **read only** options to TRUE.

After the database is loaded, SQL Server may need to complete two more tasks:

- It must "zero" all unused pages, if the database being loaded into is larger than the dumped database.
- It must complete recovery, applying transaction log changes to the data.

Depending on the number of unallocated pages, or the number of long transactions, this can take a few seconds, or many hours for a very large database. SQL Server will issue messages telling you that it is zeroing pages, or that it has begun recovery. These messages are normally buffered; in order to see them, issue this command:

```
set flushmessage on
```

Applying Changes to the Database: *load transaction*

Once you have loaded the database, use the **load transaction** command (or its abbreviation, **load tran**) to load each transaction log dump in the order in which it was made. This process reconstructs the database by re-executing the changes recorded in the transaction log.

➤ ***Note***

If users have made any changes between the **load database** and **load transaction** commands, the **load transaction** fails. Lock the database, then reload it and re-apply the transaction log dumps.

When the entire sequence of transaction log dumps has been loaded, the database reflects all transactions that had committed at the time of the last transaction log dump. Use **sp_dboption** to reset the **no chkpt on recovery, dbo use only,** and **read only** options to FALSE so that users can resume making changes to the database.

Using the Special *dump transaction* Options

Under certain circumstances, the simple model described above does not apply. Table 7-2 describes when to use the special **with no_log** and **with truncate_only** options instead of the standard **dump transaction** command.

When	Use
The log is on the same segment as the data	**dump transaction with truncate_only** to truncate the log **dump database** to copy the entire database, including the log
You are not concerned with the recovery of recent transactions (for example, in an early development environment)	**dump transaction with truncate_only** to truncate the log and **dump database** to copy the entire database
Your usual method of dumping the transaction log (either the standard **dump transaction** command or **dump transaction with truncate_only**) fails because of insufficient log space	**dump transaction with no_log** to truncate the log without recording the event and **dump database** immediately after to copy the entire database, including the log

Table 7-2: When to Use dump transaction with truncate_only or with no_log

Using the Special Load Options to Identify Dump Files

Use the special **with headeronly** option to provide header information for a specified file or for the first file on a tape. Use the **with listonly** option to return information about all files on a tape.

➤ ***Note***

These options do not actually load databases or transaction logs on the tape.

Backup and Recovery Illustrated

Figure 7-1 illustrates the process of restoring a database that is created at 4:30 pm on Monday and dumped immediately after. Full database dumps are made every night at 5 pm. Transaction log dumps are made at 10 am, noon, 2 pm, and 4 pm every day:

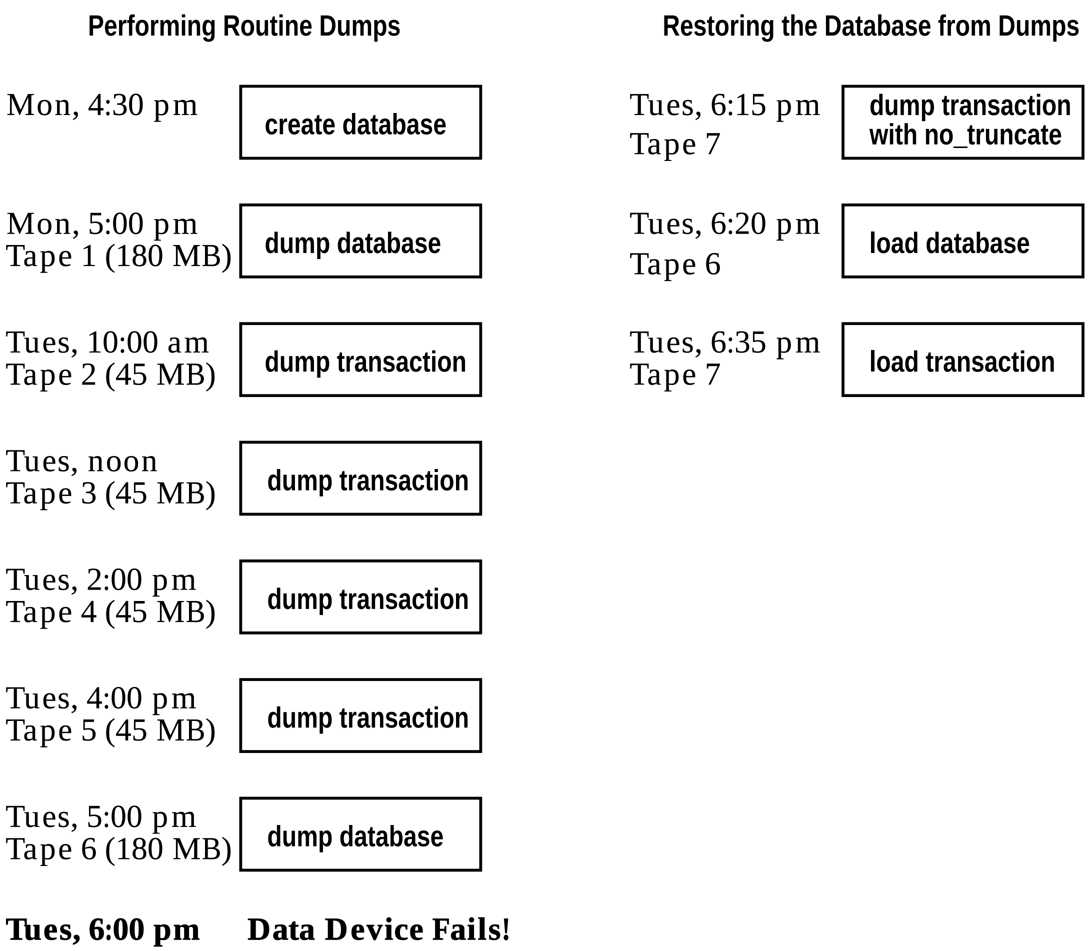


Figure 7-1: Restoring a Database from Backups

If the disk that stores the data fails on Tuesday at 6:00 pm, use the following steps to restore the database:

1. Use **dump transaction with no_truncate** to get a current transaction log dump.
2. Use **load database** to load the most recent database dump, Tape 6.
3. Use **load transaction** to apply the most recent transaction log dump, Tape 7.

Figure 7-2 illustrates how to restore the database when the data device fails at 4:59 pm on Tuesday—just before the operator is scheduled to make the nightly database dump:

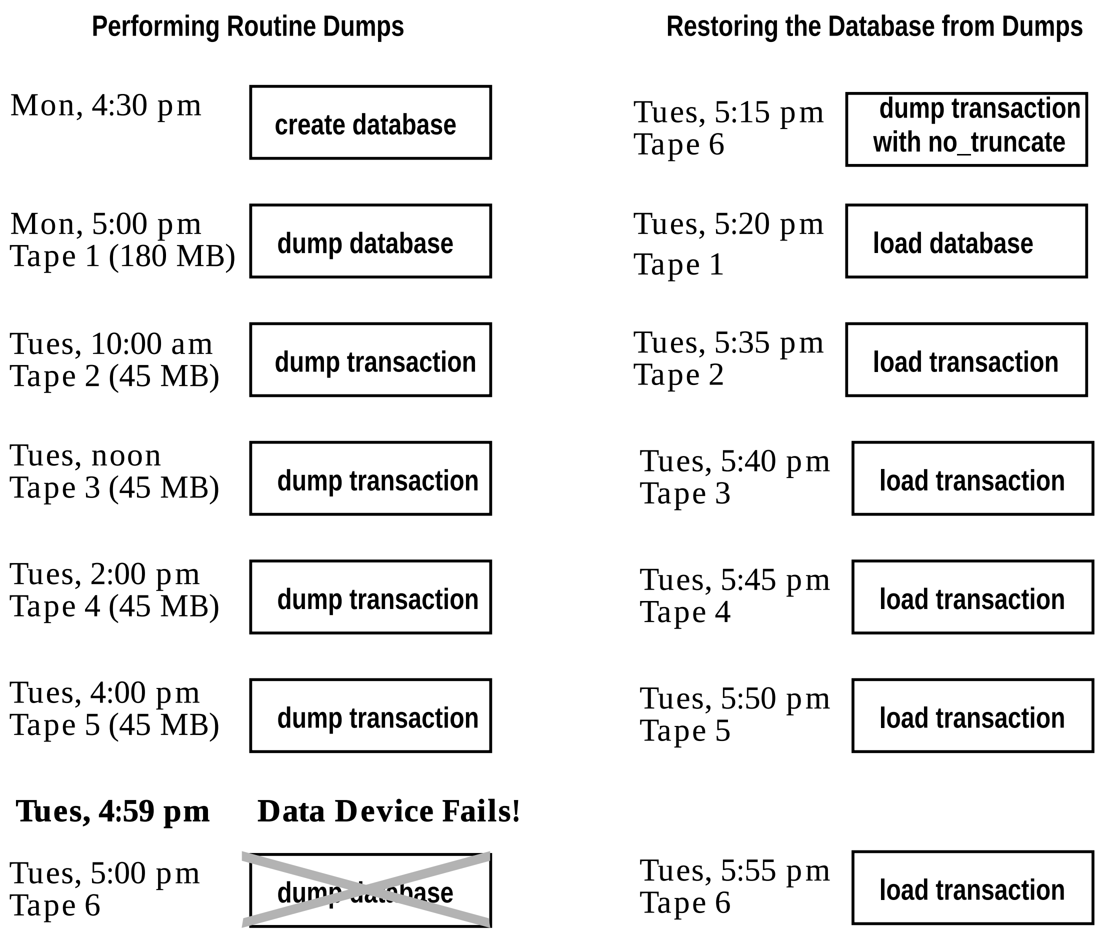


Figure 7-2: Restoring a Database, a Second Scenario

Use the following steps to restore the database:

1. Use **dump transaction with no_truncate** to get a current transaction log dump on Tape 6 (the tape you would have used for the routine database dump).
2. Use **load database** to load the most recent database dump, Tape 1.
3. Use **load transaction** to load Tapes 2, 3, 4, and 5 and the most recent transaction log dump, Tape 6.

Designating Responsibility for Backups

Many organizations have an operator whose job is to perform all backup and recovery operations. Only a System Administrator, Database Owner, or Operator can execute the dump and load commands. The Database Owner can dump only his or her own database. The Operator and System Administrator can dump and load any database.

Any user can execute **sp_volchanged** to notify the Backup Server when a tape volume is changed. On OpenVMS systems, the operator responsible for changing tape volumes must have permission to execute the REPLY command.

Using the Backup Server for Backup and Recovery

Dumps and loads are performed by an Open Server program, Backup Server, running on the same machine as SQL Server. (On VAX clusters, Backup Server can run on any machine in the cluster with access to every database device to be dumped or loaded.) You can perform backups over the network, using a Backup Server on a remote computer and another on the local computer.

Backup Server:

- Creates and loads from “striped dumps”. **Dump striping** allows you to use up to 32 backup devices in parallel. This splits the database into approximately equal portions and backs up each portion to a separate device.
- Creates and loads single dumps that span several tapes.
- Dumps and loads across the network to a Backup Server running on another machine.
- Dumps several databases or transaction logs to a single tape.

- Loads a single file from a tape that contains many database or log dumps.
- Supports platform-specific tape handling options.
- Directs volume-handling requests to the session where the dump or load command was issued, or to its operator console.
- Detects the physical characteristics of the dump devices to determine protocols, block sizes, and other characteristics.

Relationship Between SQL Server and Backup Servers

Figure 7-3 shows two users performing backup activities simultaneously on two databases:

- User1 is dumping database *db1* to a remote Backup Server.
- User2 is loading database *db2* from the local Backup Server.

Each issues the appropriate dump or load command from a SQL Server session. SQL Server interprets the command and sends remote procedure calls (RPCs) to the Backup Server. The calls indicate which database pages to dump or load, which dump devices to use, and other options.

While the dumps and loads execute, SQL Server and Backup Server use remote procedure calls to exchange instructions and status messages. Backup Server—not SQL Server—performs all data transfer for the dump and load commands.

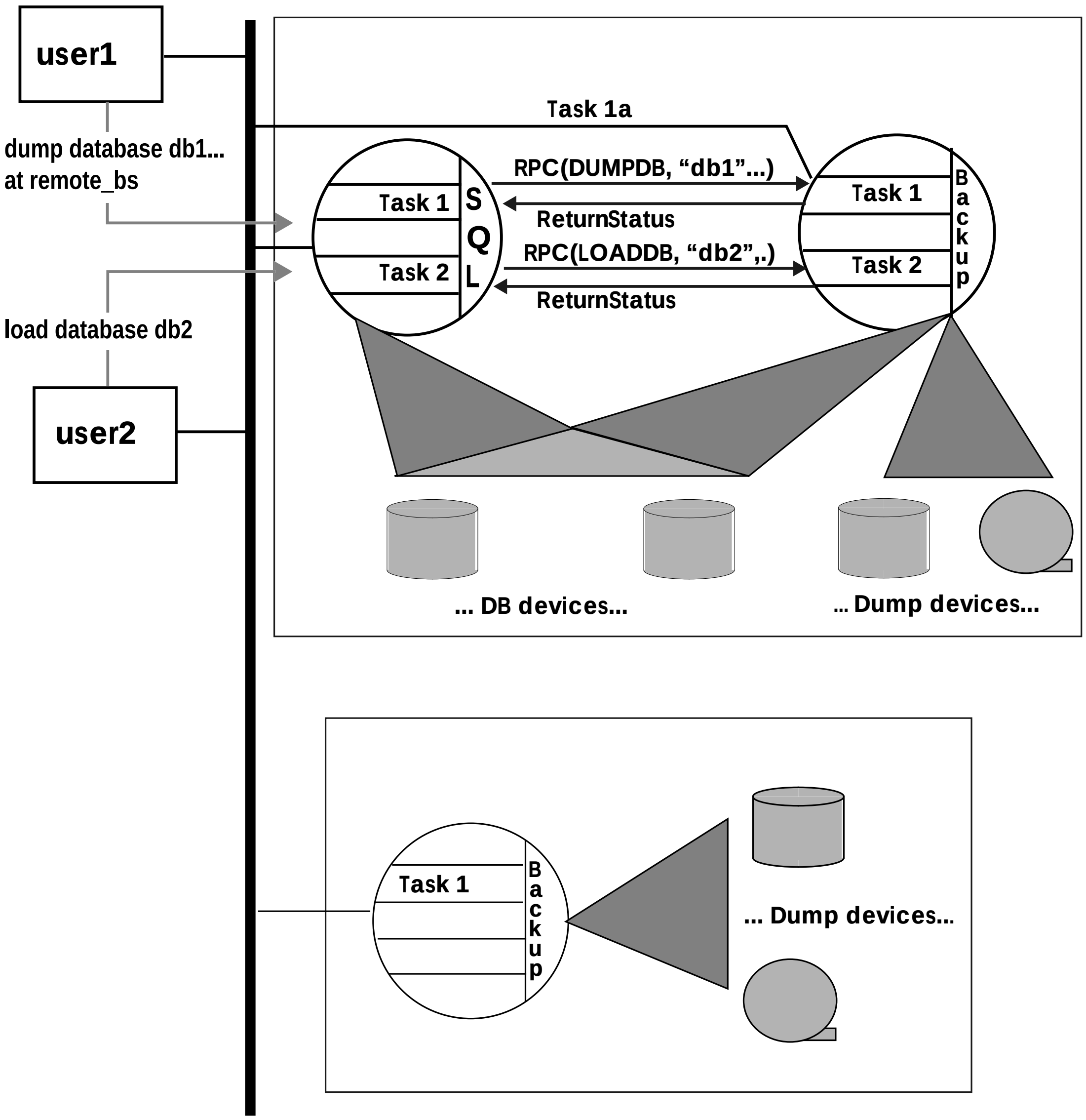


Figure 7-3: SQL Server/Backup Server Installation with a Remote Backup Server

When the local Backup Server receives user1’s dump instructions, it reads the specified pages from the database devices and sends them to the remote Backup Server. The remote Backup Server saves the data to off-line media.

Simultaneously, the local Backup Server performs user2’s load command by reading data from local dump devices and writing it to the database device.

Communicating with the Backup Server

To use the dump and load commands, your SQL Server must be able to communicate with its Backup Server. These are the requirements:

- You must have a Backup Server running on the same machine as the SQL Server (or on the same cluster for OpenVMS).
- The Backup Server must be listed in the *master..sysservers* table. The Backup Server entry, SYB_BACKUP, is created in *sysservers* when you install SQL Server. Use **sp_helpserver** to see this information. For information about changing the default Backup Server, see Chapter 12, "Managing Remote Servers".
- The Backup Server must be listed in the interfaces file. The entry for the local Backup Server is created when you install SQL Server. If you have installed a remote Backup Server on another machine, create the interfaces file on a file system shared by both machines or copy the entry to your local interfaces file.
- The user who starts the Backup Server process must have write permission for the dump devices. The "sybase" user, who usually starts SQL Server and Backup Server, can read from and write to the database devices.
- Your SQL Server must be configured for remote access. By default, SQL Server is installed with remote access enabled. See "Configuring Your Server for Remote Access" on page 7-17 for more information.

Mounting a New Volume

During the backup and restore process, it may be necessary to change tape 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 SQL Server.

On UNIX systems, the Backup Server requests a volume change when the tape capacity has been reached. The operator mounts another tape, then executes **sp_volchanged**. Figure 7-4 illustrates this process.

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. The operator uses the REPLY command to reply to these messages.

Time	**Operator, using isql**	**SQL Server**	**Backup Server**
	Issues the **dump database** command		
		Sends dump request to Backup Server	
			Receives dump request message from SQL Server Sends message for tape mounting to operator Waits for operator's reply
	Receives volume change request from Backup Server Mounts tapes Executes **sp_volchanged**		
			Checks tapes If tapes are okay, begins dump When tape is full, sends volume change request to operator
	Receives volume change request from Backup Server Mounts tapes Executes **sp_volchanged**		
			Continues dump When dump is complete, sends messages to operator and SQL Server
	Receives message that dump is complete Removes and labels tapes	Receives message that dump is complete Releases locks Completes the **dump database** command	

Figure 7-4: Changing Tape Volumes on a UNIX System

Starting and Stopping the Backup Server

Use the **startserver** utility to start a Backup Server on the same machine as the SQL Server. For more information about **startserver** syntax, see the *SQL Server Utility Programs* for your operating system.

Use the **shutdown** command to shut down a Backup Server. See“Diagnosing System Problems” and Volume 1 of the *SQL Server Reference Manual* for information about this command.

Configuring Your Server for Remote Access

The **remote access** configuration variable is set to 1 when you install SQL Server. This allows your SQL Server to execute remote procedure calls to the Backup Server.

For security reasons, you may want to disable remote access except during times when dumps and loads are taking place. Use the following commands to disable remote access:

```
sp_configure "remote access", 0
reconfigure
```

Use the following commands to re-enable remote access right before a dump or load:

```
sp_configure "remote access", 1
reconfigure
```

The **remote access** configuration variable is dynamic, and does not require a reboot of SQL Server to take effect. Only a System Security Officer can set the **remote access** variable. Only a System Administrator can execute the **reconfigure** command.

Choosing Backup Media

Tapes are preferred as dump devices since they permit a library of database and transaction log dumps to be kept off-line. Large databases can span multiple tape volumes. On UNIX systems, the Backup Server requires non-rewinding tape devices for all dumps and loads.

For a list of supported dump devices, see the *System Administration Guide Supplement* for your platform.

Protecting Backup Tapes From Being Overwritten

The **tape retention** configuration variable determines how many days backup tapes are protected from being overwritten. When you install SQL Server, **tape retention** has a value of 0. This means that backup tapes can be overwritten immediately.

Use **sp_configure** to change the **tape retention** value. The new value takes effect the next time you reboot SQL Server:

```
sp_configure "tape retention", 14

reconfigure
```

Both the **dump database** and **dump transaction** commands provide a **retaindays** option, which overrides the **tape retention** value for a particular dump.

Dumping to Files or Disks

In general, dumping to a file or to a disk is not recommended. If the disk or computer containing that file crashes, there may be no way to recover the dumps. On UNIX and PC systems, this is your only option if the *master* database is too large to fit on a single tape volume, unless you have a second SQL Server that can issue **sp_volchanged** requests.

Dumps to a file or disk can be copied to tape for off-line storage, but these tapes must be copied back to an on-line file before they can be read by SQL Server. A dump that is made to a disk file and then copied to tape cannot be read directly from tape by Backup Server.

Creating Logical Device Names for Local Dump Devices

If you are dumping to or loading from local devices (that is, you aren't performing backups across a network to a remote Backup Server), you can specify dump devices either by providing their physical locations or by specifying their logical device names. In this case, you may want to create logical dump device names in the *sysdevices* system table of the *master* database.

➤ ***Note***

If you are dumping to or loading from a remote Backup Server, you must specify the absolute pathname of the dump device. You cannot use a logical device name.

The *sysdevices* table stores information about each database and backup device, including its *physical_name* (the actual operating system device or file name) and its *device_name* (or logical name, known only within SQL Server). On most platforms, SQL Server has one or two aliases for tape devices installed in *sysdevices*. The physical names for these devices are common disk drive names for the platform; the logical names are *tapedump1* and *tapedump2*.

When creating backup scripts and threshold procedures, it is advantageous to use logical names, rather than physical device names, whenever possible. Scripts and procedures that refer to actual device names must be modified each time a backup device is replaced. If your scripts and procedures refer to logical device names, you can simply drop the *sysdevices* entry for the failed device and create a new entry that associates the logical name with a different physical device.

Listing the Current Device Names

To list the backup devices for your system, run the following query:

```
select * from master..sysdevices
  where status = 16 or status = 24
```

To list both the physical and logical names for database and backup devices, use the **sp_helpdevice** system procedure, as below:

```
sp_helpdevice tapedump1

device_name physical_name
 description
 status cntrltype device_number low      high
 ------ --------- ------------- -------- -------
tapedump1 /dev/nrmt4

tape, 625 MB, dump device
   16         3             0        0   20000
```

Adding a Backup Device

Use the system procedure **sp_addumpdevice** to add a backup device:

```
sp_addumpdevice{ "tape" | "disk"} , device_name,
   physicalname, size
```

The *physicalname* can be either an absolute pathname or a relative pathname. During dumps and loads, the Backup Server resolves relative pathnames by looking in SQL Server's current working directory.

The *size* is the capacity of the tape in megabytes. OpenVMS systems ignore the *size* parameter if it is specified. Other platforms require this parameter for tape devices but ignore it for disk devices. The Backup Server uses the *size* parameter if the dump command does not specify a tape capacity.

The *size* must be at least five database pages (each page requires 2 megabytes for most platforms, 4 megabytes for Stratus) and should be slightly below the rated capacity for the device.

Redefining a Logical Device Name

To use an existing logical device name for a different physical device, drop the device with **sp_dropdevice** and then add it with **sp_addumpdevice**. For example:

```
sp_dropdevice tapedump2

sp_addumpdevice "tape", tapedump2, "/dev/nrmt8", 625
```

Scheduling Backups of User Databases

One of the major tasks in developing a backup plan is to determine how often to back up your databases. The frequency of your backups determines how much work you can lose in the event of a media failure. This section presents some guidelines about when to dump user databases and transaction logs.

Scheduling Routine Backups

Dump each user database just after you create it, to provide a base point, and on a fixed schedule thereafter. Daily backups of the transaction log and weekly database backups are the minimum recommended. Many installations with large and active databases make database dumps every day and transaction log dumps every half hour or hour.

Interdependent databases—databases where there are cross-database transactions, triggers, or referential integrity—should be backed up at the same time, during a period when there is no cross-database data modification activity. If one of these databases fails and needs to be reloaded, they should all be reloaded from these simultaneous dumps.

◆ ***WARNING!***

Always dump both databases immediately after adding, changing, or removing a cross-database constraint or dropping a table that contains a cross-database constraint.

Other Times to Back Up a Database

In addition to routine dumps, you should dump a database each time you create a new index, perform an unlogged operation, or run the **dump transaction with no_log** or **dump transaction with truncate_only.**

Dumping a Database After Creating an Index

When you add an index to a table, the **create index** command is recorded in the transaction log. As it fills the index pages with information, however, SQL Server does not log the changes.

If your database device fails after you create an index, the **load transaction** command may take as long to reconstruct the index as the **create index** command took to build it. To avoid lengthy delays, dump each database immediately after creating an index on one of its tables.

Dumping A Database After Unlogged Operations

SQL Server writes the data for the following commands directly to disk:

- Non-logged **writetext**
- **select into** on a permanent table
- “Fast” bulk copy (**bcp** into a table with no triggers or indexes)

Because these changes are not recorded in the transaction log, you cannot recover them from a transaction log dump or recover any changes made to the database after one of these commands. To ensure that these commands are recoverable, issue a **dump database** command immediately after executing any of these operations.

Dumping a Database When the Log Has Been Truncated

The **dump transaction with truncate_only** and **dump transaction with no_log** commands remove transactions from the log without making a backup copy. To ensure recoverability, you must dump the database each time you are forced to run either command by lack of disk space. SQL Server prohibits you from copying the transaction log until you have done so.

If the **trunc log on chkpt** database option is set to true, SQL Server automatically truncates the log each time an automatic checkpoint occurs. If this option is set on, you must dump the entire database—not the transaction log—to ensure recoverability.

Scheduling Backups of *master*

Backups of the *master* database are used as part of the recovery procedure in case of a failure that affects the *master* database. If you don't have a current backup of *master*, you may be forced to reconstruct vital system tables at a time when you are under pressure to get your databases up and running again. Be prepared; back up the *master* database **regularly and frequently**.

Dumping *master* After Each Change

Back up the *master* database with **dump database** each time you change it. Although you can restrict the creation of database objects in *master*, system procedures such as **sp_addlogin** and **sp_droplogin, sp_password** and **sp_modifylogin** allow users to modify its system tables. Be sure to back up the *master* database on a frequent basis to record these changes.

Back up the *master* database after each command that affects disks, storage, databases, or segments. Always back up *master* after issuing any of the following commands and system procedures:

- **disk init**, **sp_addumpdevice** and **sp_dropdevice**
- All disk mirroring commands
- The segment system procedures **sp_addsegment, sp_dropsegment** and **sp_extendsegment**

- **create procedure** and **drop procedure**
- **sp_logdevice**
- **sp_configure**
- **create database** and **alter database**

Saving Scripts and System Tables

For further protection, save the scripts containing all of your **disk init**, **create database**, and **alter database** commands and make a hardcopy of your *sysdatabases*, *sysusages*, and *sysdevices* tables each time you issue one of these commands.

Changes that result from these commands cannot be recovered automatically with **buildmaster.** If you keep your scripts—files containing Transact-SQL statements—you can run them to recreate the changes. Otherwise, you must re-issue each command against the rebuilt master database.

You should also keep a hardcopy of the *syslogins*. When you recover *master* from a dump, compare the hardcopy to your current version of the table to be sure that users retain the same user IDs.

Truncating *master*'s Transaction Log

Since the *master* database's transaction log is on the same database devices as the data, you cannot back up its transaction log separately. You cannot move the log of the *master* database. You must always use **dump database** to back up the master database. Use **dump transaction** with the **truncate_only** option periodically (for instance, after each database dump) to purge the transaction log of the *master* database.

Scheduling Backups of *model*

Keep a current database dump of the *model* database. Each time you change *model*, make a new backup. If *model* is damaged and you do not have a backup, you must re-do all of the changes you have made in order to restore *model*.

Truncating *model*'s Transaction Log

model, like *master*, stores its transaction log on the same database devices as the data. You must always use **dump database** to back up the *model* database and **dump transaction** with the **truncate_only** option to purge the transaction log after each database dump.

Scheduling Backups of *sybsystemprocs*

The sybsystemprocs database stores only system procedures. It is very easy to restore this database by running the **installmaster** script, unless you make changes to the database.

If you change permissions on some system procedures, or create your own system procedures in *sybsystemprocs*, your two recovery choices are:

- Run the **installmaster** script, and then re-do all of your changes by re-creating your procedures, or re-executing your **grant** and **revoke** commands.
- Back up *sybsystemprocs* each time your change it.

Both of these recovery options are described in Chapter 9, "Backing Up and Restoring the System Databases".

Like other system databases, *sybsystemprocs* stores its transaction log on the same device as the data. You must always use **dump database** to back up the *sybsystemprocs* database. By default, the **trunc log on chkpt** option is turned on in *sybsystemprocs*, so you should not need to truncate the transaction log. If you change this database option, be sure to truncate the log when you dump the database.

If you are running on a UNIX system or on a PC and have only one SQL Server that can communicate with your Backup Server, be sure that the entire dump of *sybsystemprocs* fits on a single dump device. Signalling volume changes requires the **sp_volchanged** system procedure, and you cannot use this procedure on a server while the *sybsystemprocs* database is in the process of recovery.

Gathering Backup Statistics

Once you have finished reading this chapter, you should read Chapter 8, "Backing Up and Restoring User Databases" and Chapter 9, "Backing Up and Restoring the System Databases". Then practice using the backup and load commands.

Use **dump database** to make several practice backups of an actual user database and **dump transaction** to back up a transaction log. Recover the database with the **load database** command and apply successive transaction log dumps with the **load transaction** command.

Keep statistics on how long each dump and load takes and how much space it requires.The more closely you approximate real-life backup conditions, the more meaningful your predictions will be.

Once you have developed and tested your backup procedures, commit them to paper. Determine a reasonable backup schedule and adhere to it. If you develop, document, and test your backup procedures ahead of time, you will be much better prepared to get your databases online when disaster strikes.

8 Backing Up and Restoring User Databases

Introduction

Regular and frequent backups are your only protection against database damage that results from failure of your database devices.

This chapter is part of a four-chapter unit on backup and recovery. It describes how to use the dump and load commands for backup, recovery, and log truncation. It includes information on command syntax and step-by-step procedures for recovering databases after a device failure.

For more information about	See
Backup and recovery issues to address before production	Chapter 7, "Developing a Backup and Recovery Plan"
Backing up and restoring the system databases	Chapter 9, "Backing Up and Restoring the System Databases"
Using thresholds to automate backups	Chapter 10, "Managing Free Space with Thresholds"

Table 8-1: Further Information About Backup and Recovery

Dump and Load Command Syntax

The **dump database**, **dump transaction**, **load database**, and **load transaction** commands have parallel syntax. Routine dumps and loads require the name of a database and at least one dump device. The commands can also include the following options:

- **at** *server_name* to specify the remote Backup Server
- **density**, **blocksize**, and **capacity** to specify tape storage characteristics
- **dumpvolume** to specify the volume name of the ANSI tape label
- **file =** *filename* to specify the name of the file to dump to or load from
- **stripe on** *stripe_device* to specify additional dump devices
- **dismount**, **unload**, **init**, and **retaindays** to specify tape handling
- **notify** to specify whether Backup Server messages are sent to the **client** that initiated the dump or load or to the **operator_console**

Table 8-4 shows the syntax for routine database and log dumps and for dumping the log after a device failure. It indicates what type of information each part of the **dump database** and **dump transaction** statement provides:

Information Provided	Task	
	Routine Database or Log Dump	**Log Dump After Device Failure**
Command	dump {database | transaction}	dump transaction
Database name	*database_name*	*database_name*
Dump device	to *stripe_device*	to *stripe_device*
Remote Backup Server	[at *server_name*]	[at *server_name*]
Tape device characteristics	[density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*]	[density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*]
Volume name	[, dumpvolume = *volume_name*]	[, dumpvolume = *volume_name*]
File name	[, file = *file_name*]	[, file = *file_name*]
Characteristics of additional devices (up to 31 devices; one set per device)	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, file = *file_name*, dumpvolume = *volume_name*]] ...	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes,* file = *file_name,* dumpvolume = *volume_name*]] ...
Options that apply to entire dump	[with { density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, file = *file_name*, [nodismount | dismount], [nounload | unload], [retaindays = *number_days*], [noinit | init]**,** file = *file_name***,** dumpvolume = *volume_name*	[with { density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, file = *file_name*, [nodismount | dismount], [nounload | unload], [retaindays = *number_days*], [noinit | init]**,** file = *file_name***,** dumpvolume = *volume_name***,**
Do not truncate log		no_truncate
Message destination	[, notify = {client | operator_console}] }]	[, notify = {client | operator_console}] }]

Table 8-2: Syntax for Routine Dumps and for Log Dumps After Device Failure

Table 8-3 shows the syntax for loading a database, applying transactions from the log, and returning information about dump headers and files:

Information Provided	**Task**	
	Load Database or Apply Recent Transactions	**Return Header or File Information But Do Not Load Backup**
Command	load {database | transaction}	load {database | transaction}
Database name	*database_name*	*database_name*
Dump device	*from stripe_device*	*from stripe_device*
Remote Backup Server	[at *server_name*]	[at *server_name*]
Tape device characteristics	[density = *density*, blocksize = *number_bytes*]	[density = *density*, blocksize = *number_bytes*]
Volume name	[, dumpvolume = *volume_name*]	[, dumpvolume = *volume_name*]
File name	[, file = *file_name*]	[, file = *file_name*]
Characteristics of additional devices (up to 31 devices; one set per device)	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, file = *file_name*, dumpvolume = *volume_name*]] ...	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, file = *file_name*, dumpvolume=*volume_name*]] ...
Tape handling	[with{ [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload]	[with{ [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload]
Provide header information		[, headeronly]
List dump files		[, listonly [= full]]
Message destination	[, notify = {client | operator_console}] }]	[, notify = {client | operator_console}] }]

Table 8-3: Syntax for the Load Commands

Table 8-4 shows the syntax for truncating a log:

- That is not on a separate segment
- Without making a backup copy
- With insufficient free space to successfully complete a **dump transaction** or **dump transaction with truncate_only** command.

Information Provided	Task		
	Truncate Log On Same Segment as Data	**Truncate Log Without Making a Copy**	**Truncate Log with Insufficient Free Space**
Command	dump transaction	dump transaction	dump transaction
Database name	*database_name*	*database_name*	*database_name*
Do Not Copy Log	with truncate_only	with truncate_only	with no_log

Table 8-4: Special dump transaction Options

The remainder of this chapter provides greater detail about the information specified in dump and load commands and volume change messages. Routine dumps and loads are described first, followed by log dumps after device failure and the special syntax for truncating logs without making a backup copy.

For information about the permissions required to execute the dump and load commands, refer to “Designating Responsibility for Backups” in Chapter 7, “Developing a Backup and Recovery Plan”.

Specifying the Database and Dump Device

At minimum, all dump and load commands must include the name of the database being dumped or loaded. Commands that dump or load data (rather than just truncating a transaction log) must also include a dump device:

	Backing Up a Database or Log	Loading a Database or Log
Database name **Dump device**	dump {database | tran} *database_name* to *stripe_device*	load {database | tran} *database_name* from *stripe_device*
	[at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init], [notify = {client | operator_console}] }]	[at *server_name*] [density = *density*, blocksize = *number_bytes* dumpvolume = *volume_name* file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], [notify = {client | operator_console}] }]

Table 8-5: Indicating the Database Name and Dump Device

Rules for Specifying Database Names

The database name can be specified as a literal, a local variable, or a parameter to a stored procedure.

If you are loading a database from a dump:

- The database must exist. You can create a database with the **for load** option of **create database**, or load over an existing database. Loading a database always over-writes all of the information in the existing database.

- You do not need to give the same database name as the name of the database you dumped. For example, you can dump the *pubs2* database, create another database called *pubs2_archive*, and load the dump into the new database.

◆ ***WARNING!***

You should never change the name a database that contains primary keys for references from other databases. If you must load a dump from such a database and provide a different name, first drop the references to it from other databases.

Rules for Specifying Dump Devices

Use the following rules when specifying the dump device:

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You cannot dump to or load from the "null device" (on UNIX, */dev/null*; on OpenVMS, any device name beginning with NL; not applicable to PC platforms).
- When dumping to or loading from a local device, you can use any of the following forms to specify the dump device:
  - An absolute pathname
  - A relative pathname
  - A logical device name from the *sysdevices* system table

  The Backup Server resolves relative pathnames using SQL Server's current working directory.
- When dumping or loading across the network:
  - You must specify the absolute pathname of the dump device. (You cannot use a relative pathname or a logical device name from the *sysdevices* system table.)
  - The pathname 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 (_), you must enclose it in quotes.

Examples

The following examples use a single device for dumps and loads. (It is not necessary to use the same device for dumps and loads.)

On UNIX:

```
dump database pubs2 to "/dev/nrmt4"
```

```
load database pubs2 from "/dev/nrmt4"
```

On OpenVMS:

```
dump database pubs2 to "MTA0:"
```

```
load database pubs2 from "MTA0:"
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Specifying a Remote Backup Server

Use the **at** *server_name* clause to send dump and load requests across the network to a Backup Server running on another machine:

	Backing Up a Database or Log	Loading a Database or Log
	dump {database | tran} *database_name* to *stripe_device*	load {database | tran} *database_name* from *stripe_device*
Remote Backup Server	[at *server_name*]	[at *server_name*]
	[density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init], [notify = {client | operator_console}] }]	[density = *density*, blocksize = *number_bytes,* dumpvolume = *volume_name,* file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], [notify = {client | operator_console}] }]

Table 8-6: Dumping to or Loading from a Remote Backup Server

This is ideal for installations that use a single machine with multiple tape devices for all backups and loads. Operators can be stationed at these machines, ready to service all tape change requests.

The *server_name* must appear in the interfaces file on the computer where SQL Server is running, but does not need to appear in the *sysservers* table.

The following examples dump to and load from the remote Backup Server REMOTE_BKP_SERVER:

On UNIX:

```
dump database pubs2
        to "/dev/nrmt0" at REMOTE_BKP_SERVER
```

```
load database pubs2
        from "/dev/nrmt0" at REMOTE_BKP_SERVER
```

On OpenVMS:

```
dump database pubs2
    to "MTA0:" at REMOTE_BKP_SERVER
```

```
load database pubs2
    from "MTA0:" at REMOTE_BKP_SERVER
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Specifying Tape Density, Blocksize, and Capacity

In most cases, the Backup Server uses a default tape density and blocksize that are optimal for your operating system; **we recommend that you use them**. The **density** and **blocksize** options allow you to override these defaults when they are not appropriate for particular devices. The **capacity** option allows you to specify tape capacity on platforms that do not reliably detect the end-of-tape marker.

You can specify a density, blocksize, and capacity for each dump device. You can also specify the **density, blocksize**, and **capacity** options in the **with** clause, for all dump devices. Characteristics that are specified for an individual tape device take precedence over those specified in the **with** clause.

	Backing Up a Database or Log	Loading a Database or Log
	dump {database | tran} *database_name* to *stripe_device* [at *server_name*]	load {database | tran} *database_name* from *stripe_device* [at *server_name*]
Characteristics of a Single Tape Device	[density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*,	[density = *density*, blocksize = *number_bytes*,
	dumpvolume = *volume_name*, file = file_name] [stripe on *stripe_device*] [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = file_name] ...]	dumpvolume = *volume_name*, file = *file_name*] [stripe on *stripe_device*] [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...]
Characteristics of All Dump Devices	[with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*,	[with{ density = *density*, blocksize = *number_bytes*,
	dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init], [notify = {client | operator_console}] }]	dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], [notify = {client | operator_console}] }]

Table 8-7: Specifying Tape Density, Blocksize, and Capacity

The following sections provide greater detail about the **density, blocksize**, and **capacity** options.

Overriding the Default Density

The dump and load commands use the default tape density for your operating system. In most cases, this is the optimal density for tape dumps.

When dumping to tape on OpenVMS systems, you can override the default density with the **density** = *density* option. Valid densities are 800, 1600, 6250, 6666, 10000, and 38000. Not all densities are valid for all tape drives; specify a value that is correct for your drive.

This option has no effect on OpenVMS tape loads or on UNIX and PC platform dumps or loads.

➤ ***Note***

Specify tape density only when using the **init** tape handling option. For more information on this option, see the section “Reinitializing a Volume Before a Dump” in this chapter.

Overriding the Default Blocksize

By default, the dump and load commands choose the “best” blocksize for your operating system. Wherever possible, use these defaults.

You can use the **blocksize** = *number_bytes* option to override the default blocksize for a particular dump device. The blocksize must be at least one database page (2048 bytes for most systems, 4096 for Stratus), and must be an exact multiple of the database page size.

For OpenVMS systems, you can specify a blocksize only for dumps. Use a blocksize less than or equal to 16384.

For UNIX systems, you can specify a blocksize on both dump and load commands. When loading a dump, you must specify the same blocksize that was used to make the dump.

Specifying Tape Capacity for Dump Commands

By default, OpenVMS systems write until they reach the physical end-of-tape marker, then signal that a volume change is required. For UNIX platforms that cannot reliably detect the end-of-tape marker, you must indicate how many kilobytes can be dumped to a tape.

If you specify the physical pathname of the dump device, you must include the **capacity** = *number_kilobytes* parameter in the dump command. If you specify the logical dump device name, the Backup Server uses the *size* parameter stored in the *sysdevices* system table unless you override it with the **capacity** = *number_kilobytes* parameter.

The specified capacity must be at least five database pages (each page requires 2048 bytes for most platforms, 4096 for Stratus). We recommend that you specify a capacity that is slightly below the rated capacity for your device.

Specifying the Volume Name

Use the **with dumpvolume =** *volume_name* option to specify the volume name. The **dump database** and **dump transaction** commands write the volume name to the SQL tape label. The **load database** and **load transaction** commands check the label; if the wrong volume is loaded, Backup Server generates an error message.

You can specify a volume name for each dump device. You can also specify a volume name in the **with** clause for all devices. Volume names specified for individual devices take precedence over those specified in the **with** clause.

	Backing Up a Database or Log	Loading a Database or Log
	dump {database | tran} *database_name* to *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*,	load {database | tran} *database_name* from *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*,
Volume name for single device	dumpvolume = *volume_name*,	dumpvolume = *volume_name,*
	file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*,	file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*]...] [with { density = *density*, blocksize = *number_bytes*,
Volume name for all devices	dumpvolume = *volume_name*,	dumpvolume = *volume_name*,
	file = *file_name*, [nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init], [notify = {client | operator_console}] }]	file = *file_name*, [nodismount | dismount], [nounload | unload], [notify = {client | operator_console}] }]

Table 8-8: Specifying the Volume Name

Identifying a Dump

When you dump a database or transaction log, Backup Server creates a default file name for the dump by concatenating the:

- Last seven characters of the database name
- Two-digit year number
- Three-digit day of the year (1 through 366)
- Number of seconds since midnight, in hexadecimal

You can override this default using the **file =** *file_name* option. The file name cannot exceed 17 characters and must conform to the file naming conventions for your operating system.

You can specify a file name for each dump device. You can also specify a file name for all devices in the **with** clause. File names specified for individual devices take precedence over those specified in the **with** clause.

	Backing Up a Database or Log	**Loading a Database or Log**
	dump {database | tran} *database_name* to *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*,	load {database | tran} *database_name* from *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*,
File name for single device	file = file_name]	file = *file_name*]
	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*,	[stripe on *stripe_device*] [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*,
File name for all devices	file = *file_name,*	file = *file_name,*
	[nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init], [notify = {client | operator_console}] }]	[nodismount | dismount], [nounload | unload], [notify = {client | operator_console}] }]

Table 8-9: Specifying the File Name for a Dump

The following examples dump the transaction log for the *publications* database without specifying a file name. The default file name, *cations930590E100*, identifies the database and the date and time the dump was made:

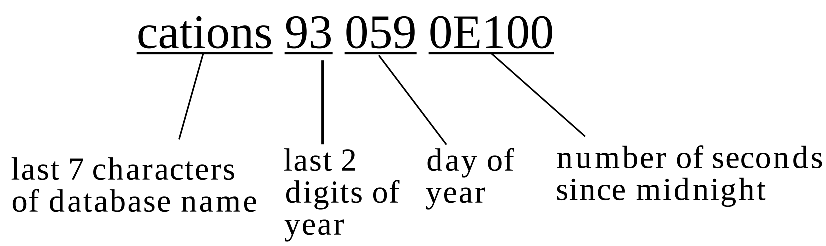


Figure 8-1: Default File Naming Convention for Database and Transaction Log Dumps

Backup Server sends the file name to the default message destination or to the **notify** location for the dump command. Be sure to label each backup tape with the volume name and file name before storing it.

When you load a database or transaction log, you can use the **file =** *file_name* clause to specify which dump to load from a volume that contains multiple dumps:

When loading the dump from a multi-file volume, you must specify the correct file name:

On UNIX:

```
dump tran publications
    to "/dev/nrmt3"
load tran publications
    from "/dev/nrmt4"
    with file = "cations930590E100"
```

On OpenVMS:

```
dump tran publications
    to "MTA01"

load database mydb
    from "MTA01:"
    with file = "cations930590E100"
```

The following examples use a user-defined file naming convention. The 15-character file name, *mydb93jul201800*, identifies the database (*mydb*), the date (July 20, 1993), and time (18:00, or 6:00 pm) the dump was made. The load command advances the tape to *mydb93jul201800* before loading:

On UNIX:

```
dump database mydb
    to "/dev/nrmt3"
    with file = "mydb93jul201800"
```

```
load database mydb
    from "/dev/nrmt4"
    with file = "mydb93jul201800"
```

On OpenVMS:

```
dump database mydb
    to "MTA01"
    with file = "mydb93jul201800"
```

```
load database mydb
    from "MTA01:"
    with file = "mydb93jul201800"
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Specifying Additional Dump Devices: the *stripe on* Clause

Dump striping allows you to use multiple dump devices for a single dump or load command. Use a separate **stripe on** clause to specify the name (and, if desired, the characteristics) of each device.

Each dump or load command can have up to 31 **stripe on** clauses (for a maximum of 32 dump devices):

	Backing Up a Database or Log	Loading a Database or Log
	dump {database | tran} *database_name* to *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*]	load {database | tran} *database_name* from *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes,* dumpvolume = *volume_name,* file = *file_name*]
Characteristics of an additional tape device (one set per device; up to 31 devices)	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name***,** file = *file_name*] ...]	[stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...]
	[with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init], [notify = {client | operator_console}] }]	[with{ density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], [notify = {client | operator_console}] }]

Table 8-10: Using More Than One Dump Device

Dumping to Multiple Devices

The Backup Server divides 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 dump an individual database or transaction log. Because each tape stores just a portion of the database, it is less likely that new tapes must be mounted on a particular device.

Loading from Multiple Devices

You can use up to 32 devices to load a database or transaction log. Using multiple devices decreases both the time required for the load and the likelihood of having to mount multiple tapes on a particular device.

Using Fewer Devices to Load than to Dump

You can load a database or log even if one of your dump devices becomes unavailable between the dump and load. Specify fewer stripe clauses in the load command than you did in the dump command.

➤ ***Note***

When you dump and load across the network, you must use the same number of drives for both operations.

The following examples use three devices to dump a database but only two to load it:

On UNIX:

```
dump database pubs2 to “"dev/nrmt0"
    stripe on "/dev/nrmt1"
    stripe on "/dev/nrmt2"
```

```
load database pubs2 from "/dev/nrmt0"
    stripe on "/dev/nrmt1"
```

On OpenVMS:

```
dump database pubs2 to "MTA0:"
    stripe on “MTA1:”
    stripe on “MTA2:”
```

```
load database pubs2 from  "MTA0:"
    stripe on "MTA1:"
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

After the first two tapes are loaded, a message notifies the operator to load the third.

Specifying the Characteristics of Individual Devices

Use a separate **at** *server_name* clause for each stripe device attached to a remote Backup Server. If you do not specify a remote Backup Server name, the local Backup Server looks for the dump device on the local machine. If necessary, you can also specify separate tape device characteristics (**density**, **blocksize**, **capacity**, **dumpvolume**, and **file**) for individual stripe devices.

The following examples use three dump devices, each attached to the remote Backup Server REMOTE_BKP_SERVER:

On UNIX:

```
dump database pubs2
        to "/dev/nrmt0" at REMOTE_BKP_SERVER
        stripe on "/dev/nrmt1" at REMOTE_BKP_SERVER
        stripe on "/dev/nrmt2" at REMOTE_BKP_SERVER
```

On OpenVMS:

```
dump database pubs2
    to "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
    stripe on "MTA2:" at REMOTE_BKP_SERVER
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Tape Handling Options

The tape handling options, which appear in the **with** clause, apply to all devices used for the dump or load. They include:

- **nodismount** to keep the tape available for additional dumps or loads
- **unload** to rewind and unload the tape following the dump or load
- **retaindays** to protect files from being overwritten
- **init** to reinitialize the tape rather than appending the dump files after the last end-of-tape mark

	Backing Up a Database or Log	Loading a Database or Log
	dump {database | tran} *database_name* to *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*,	load {database | tran} *database_name* from *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes* dumpvolume = *volume_name* file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*,
Tape Handling Options	[nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init],	nodismount | dismount], [nounload | unload],
	[notify = {client | operator_console}] }]	[notify = {client | operator_console}] }]

Table 8-11: Tape Handling Options

Specifying Whether to Dismount the Tape

On platforms that support logical dismounts, such as OpenVMS, tapes are dismounted when a dump or load completes. Use the **nodismount** option to keep the tape mounted and available for additional dumps or loads. This command has no effect on UNIX or PC systems.

Rewinding the Tape

By default, both dump and load commands use the **nounload** tape handling option.

On UNIX systems, this prevents the tape from rewinding after the dump or load completes. This allows you to dump additional databases or logs to the same volume, or to load additional databases or logs from that volume. Use the **unload** option for the last dump on the tape, to rewind and unload the tape when the command completes.

On OpenVMS systems, tapes are always rewound after a dump or load completes. Use the **unload** option to unthread the tape and eject it from the drive. (This action is equivalent to the /UNLOAD qualifier for the OpenVMS DISMOUNT command.)

Protecting Dump Files from Being Overwritten

The **tape retention** configuration variable specifies the number of days that must elapse between the creation of a tape file and the time at which you can overwrite it with another dump. This server-wide variable, which is set with the **sp_configure** system procedure, applies to all dumps requested from a single SQL Server.

Use the **retaindays** = *number_days* option to override the **tape retention** variable for a single database or transaction log dump. The number of days must be a positive integer, or zero if the tape can be overwritten immediately.

➤ ***Note***

tape retention and **retaindays** are meaningful only for disk, quarter-inch cartridge, and single-file media. On multi-file media, the Backup Server only checks the expiration date of the first file.

Reinitializing a Volume Before a Dump

By default, each dump is appended to the tape following the last end-of-tape mark. Tape volumes are not reinitialized. This allows you to dump multiple databases to a single volume. (New dumps can be appended only to the last volume of a multi-volume dump.)

Use the **init** option to overwrite any existing contents of the tape. If you specify **init**, the Backup Server reinitializes the tape **without** checking for:

- ANSI access restrictions
- Files that have not yet expired

- Non-Sybase data ("foreign" tapes on OpenVMS)

The default, **noinit**, checks for all three conditions and sends a volume change prompt if any are present.

The following examples initialize two devices, overwriting the existing contents with the new transaction log dumps:

On UNIX:

```
dump transaction pubs2
    to "/dev/nrmt0"
    stripe on "/dev/nrmt1"
    with init
```

On OpenVMS:

```
dump transaction pubs2
    to "MTA0:"
    stripe on "MTA1:"
    with init
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Dumping Multiple Databases to a Single Volume

Follow these steps to dump multiple databases to the same tape volume:

1. Use the **init** option for the first database. This overwrites any existing dumps and places the first dump at the beginning of the tape.
2. Use the default (**noinit** and **nounload**) option for subsequent databases. This places them one after the other on the tape.
3. Use the **unload** option for the last database on the tape. This rewinds and unloads the tape after the last database is dumped.

Figure 8-2 illustrates which options to use to dump three databases to a single tape volume:

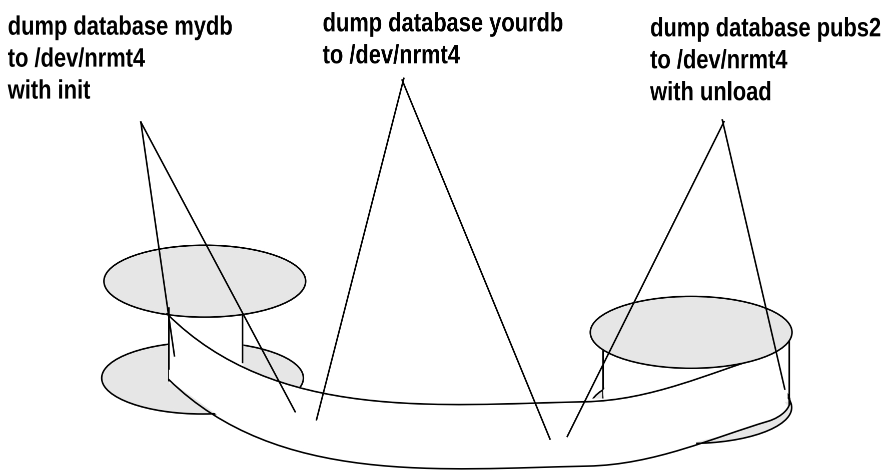


Figure 8-2: Dumping Several Databases to the Same Volume

Overriding the Default Message Destination

Backup Server messages inform the operator when to change tape volumes and how the dump or load is progressing. The default destination for these messages depends on whether the operating system offers an operator terminal feature. The **notify** option, which appears in the **with** clause, allows you to override the default message destination for a dump or load.

On operating systems, such as OpenVMS, that offer an operator terminal feature, volume change messages are always sent to an operator terminal on the machine where the Backup Server is running. (OpenVMS routes messages to terminals that are enabled for TAPES, DISKS, or CENTRAL.) Use **notify = client** to route other Backup Server messages to the terminal session where the **dump** or **load** request initiated.

On systems such as UNIX that do not offer an operator terminal feature, messages are sent to the client that initiated the dump or load request. Use **notify = operator_console** to route messages to the terminal where the remote Backup Server was started.

<table>
<tr><th></th><th>Backing Up a Database or Log</th><th>Loading a Database or Log</th></tr>
<tr><td></td><td>dump {database | tran} database_name
to stripe_device
[at server_name]
[density = density,
blocksize = number_bytes,
capacity = number_kilobytes,
dumpvolume = volume_name,
file = file_name]
[stripe on stripe_device
[at server_name]
[density = density,
blocksize = number_bytes,
capacity = number_kilobytes,
dumpvolume = volume_name,
file = file_name] ...]
[with{
density = density,
blocksize = number_bytes,
capacity = number_kilobytes,
dumpvolume = volume_name,
file = file_name,
[nodismount | dismount],
[nounload | unload],
retaindays = number_days,
[noinit | init],</td><td>load {database | tran} database_name
from stripe_device
[at server_name]
[density = density,
blocksize = number_bytes
dumpvolume = volume_name
file = file_name]
[stripe on stripe_device
[at server_name]
[density = density,
blocksize = number_bytes,
dumpvolume = volume_name,
file = file_name] ...]
[with{
density = density,
blocksize = number_bytes,
dumpvolume = volume_name,
file = file_name,
[nodismount | dismount],
[nounload | unload],</td></tr>
<tr><td>Message destination</td><td>[notify = {client | operator_console}] }]</td><td>[notify = {client | operator_console}] }]</td></tr>
</table>

Table 8-12: Overriding the Default Message Destination

Getting Information About Dump Files

If you are unsure of the contents of a tape, use the **with headeronly** or **with listonly** option of the load commands to request a report with this information.

	Listing Information About a Dump
	load {database | tran} *database_name* from *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes* dumpvolume = *volume_name* file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload],
List header information **List files on tape**	[headeronly [,file = *filename*]] , [listonly [=full]] ,
	[notify = {client | operator_console}] }]

Table 8-13: Listing Dump Headers or File Names

> ➤ ***Note***
>
> Neither **with headeronly** nor **with listonly** loads the dump files after displaying the report.

Requesting Dump Header Information

with headeronly returns the header information for a single file. If you do not specify a file name, **with headeronly** returns information about the first file on the tape.

The header indicates whether the dump is for a database or transaction log, the database ID, the file name, and the date the dump was made. For database dumps, it also shows the character set, sort order, page count and next object ID. For transaction log dumps, it shows the checkpoint location in the log, the location of the oldest **begin transaction** record and the old and new sequence dates.

The following examples return header information for the first file on the tape and then for the file *mydb9229510945*:

On UNIX:

```
load database mydb
    from "/dev/nrmt4"
    with headeronly

load database mydb
    from "/dev/nrmt4"
    with headeronly, file = "mydb9229510945"
```

On OpenVMS:

```
load database mydb
    from "MTA01:"
    with headeronly

load database mydb
    from "MTA01:"
    with headeronly, file = "mydb9229510945"
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Here is sample output from **headeronly**:

```
Backup Server session id is: 44. Use this value when executing
the ‘sp_volchanged’ system stored procedure after fulfilling any
volume change request from the Backup Server.

Backup Server: 4.28.1.1: Dumpfile name ‘mydb9232610BC8 ‘ section
number 0001 mounted on device ‘backup/SQL_SERVER/mydb.db.dump’

This is a database dump of database ID 5 from Nov 21 1992 7:02PM.

Database contains 1536 pages; checkpoint RID=(Rid pageid =
0x404; row num = 0xa); next object ID=3031; sort order ID=50,
status=0; charset ID=1.
```

Determining the Database, Device, File Name, and Date

with listonly returns a brief description of each dump file on a volume. It includes the name of the database, the device used to make the dump, the file name, the date and time the dump was made, and the date and time it can be overwritten. **with listonly = full** provides greater detail. Both reports are sorted by SQL tape label.

Following is sample output of a **load database** command **with listonly**:

```
Backup Server: 4.36.1.1: Device ‘/dev/nrst0’:
File name:‘model9320715138  ‘
Create date & time: Monday, Jul 26, 1993, 23:58:48
Expiration date & time: Monday, Jul 26, 1993, 00:00:00
Database name:‘model                         ‘
```

and sample output from **with listonly = full**:

```
Backup Server: 4.37.1.1: Device ‘/dev/nrst0’:
Label id:‘HDR1’
File name:‘model9320715138  ‘
Stripe count:0001
Device typecount:01

Archive volume number:0001
Stripe position:0000
Generation number:0001
Generation version:00
Create date & time:Monday, Jul 26, 1993, 23:58:48
Expiration date & time:Monday, Jul 26, 1993, 00:00:00
Access code:‘ ‘
File block count:000000
Sybase id string:
‘Sybase  ‘Reserved:‘       ‘

Backup Server: 4.38.1.1: Device ‘/dev/nrst0’:
Label id:‘HDR2’
Record format:‘F’
Max. bytes/block:55296
Record length:02048
Backup format version:01
Reserved:‘   ‘
Database name:‘model                         ‘
Buffer offset length:00
Reserved:‘                            ‘
```

After listing all files on a volume, the Backup Server sends a volume change request:

```
Backup Server: 6.30.1.2: Device /dev/nrst0: Volume cataloguing
complete.
Backup Server: 6.51.1.1: OPERATOR: Mount the next volume to
search.
Backup Server: 6.78.1.1: EXECUTE sp_volchanged
     @session_id = 5,
     @devname = '/dev/nrst0',
     @action = { 'PROCEED' | 'RETRY' | 'ABORT' },
     @fname = '
```

The operator can mount another volume and signal the volume change with **sp_volchanged**, or use **sp_volchanged** to terminate the search operation for all stripe devices.

Copying the Log After a Device Failure

Normally, the **dump transaction** command truncates the inactive portion of the log after copying it. Use the **with no_truncate** option to copy the log without truncating it.

Th**e no_truncate option** allows you to copy the transaction log after failure of the device that holds your data. It uses pointers in the *sysdatabases* and *sysindexes* system tables to determine the physical location of the transaction log. It can be used only if your transaction log is on a separate segment and your *master* database is accessible.

◆ ***WARNING!***

Use no_truncate only if media failure makes your data segment inaccessible. Never use no_truncate on a database that is in use.

After copying the log **with no_truncate**, you must follow the steps described in it was made. These steps are described in the section "Volume Change Prompts for Loads" in this chapter**.**

	dump transaction *database_name* to *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] [stripe on *stripe_device* [at *server_name*] [density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*] ...] [with{ density = *density*, blocksize = *number_bytes*, capacity = *number_kilobytes*, dumpvolume = *volume_name*, file = *file_name*, [nodismount | dismount], [nounload | unload], retaindays = *number_days*, [noinit | init],
Do not truncate log	no_truncate,
	[notify = {client | operator_console}] }]

Table 8-14: Copying the Log After a Device Failure

You can use **no_truncate** with striped dumps, tape initialization, and remote Backup Servers. Here is an example:

```
dump transaction mydb
to "/dev/nrmt0" at REMOTE_BKP_SERVER
with init, no_truncate,
notify = "operator_console"
```

Truncating a Log That Is Not on a Separate Segment

If a database does not have a separate log segment, you cannot use **dump transaction** to copy the log and then truncate it. For these databases, you must:

1. Use the special **with truncate_only** option of **dump transaction** to truncate the log so that it does not run out of space.
2. Use **dump database** to copy the entire database, including the log.

Because it copies no data, the **with truncate_only** option requires only the name of the database:

```
dump transaction database_name with truncate_only
```

The following example dumps the database *mydb*, which does not have a separate log segment, then truncates the log:

```
dump database mydb to mydevice
dump transaction mydb with truncate_only
```

Truncating the Log In Early Development Environments

In early development environments, the transaction log is quickly filled by the process of creating, dropping and recreating stored procedures and triggers and checking integrity constraints. Recovery of data may be less important than ensuring that there is adequate space on database devices.

The **with truncate_only** option of **dump transaction** is often useful in these environments. It allows you to truncate the transaction log without making a backup copy:

```
dump transaction database_name with truncate_only
```

After you run **dump transaction with truncate_only,** SQL Server requires you dump the database before you can run a routine log dump.

Truncating a Log That Has No Free Space

When the log is very full, you may not be able to use your usual method to dump the transaction log. If you have tried **dump transaction** or **dump transaction with truncate_only** and the command fails because of insufficient log space, use the special **with no_log** option of **dump transaction**:

```
dump transaction database_name with no_log
```

This special option truncates the log without logging the dump transaction event. Because it copies no data, it requires only the name of the database.

◆ ***WARNING!***

Use dump transaction with no_log only as a last resort.

Dangers of Using *with truncate_only* and *with no_log*

with truncate_only and **with no_log** allow you to truncate a log that has become disastrously short of free space. Neither option provides a means to recover your databases.

After running **dump transaction with truncate_only** or **with no_log**, you have no way to recover transactions that have committed since the last routine dump.

◆ ***WARNING!***

Run dump database at the earliest opportunity to ensure that your data can be recovered.

The following example truncates the transaction log for *mydb* and then dumps the database:

```
dump transaction mydb
    with no_log
dump database mydb
```

Providing Enough Log Space

Every use of **dump transaction... with no_log** is considered an error, and is recorded in the server's error log. If you have created your databases with separate log 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.

Responding to Volume Change Requests

On UNIX and PC systems, use the **sp_volchanged** system procedure to notify the Backup Server when the correct volumes have been mounted. On OpenVMS systems, use the REPLY command.

To use **sp_volchanged**, log into any SQL Server that can communicate with both the Backup Server that issued the volume change request and the SQL Server that initiated the dump or load.

sp_volchanged Syntax

Use the following syntax for **sp_volchanged**:

```
sp_volchanged session_id, device_name , action
    [ ,filename [, volume_name] ]
```

- Use the *session_id* and *device_name* parameters specified in the volume change request.
- The action specifies whether to **abort**, **proceed** with, or **retry** the dump or load.
- The *filename* specifies which file to load. If you do not specify a file name **sp_volchanged**, the Backup Server loads the **file** = *filename* parameter of the load command. If neither **sp_volchanged** nor the load command specifies which file to load, the Backup Server loads the first file on the tape.
- 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. During loads, the Backup Server uses the *volume_name* to confirm that the correct tape has been mounted. If you do not specify a *volume_name* with **sp_volchanged**, the Backup Server uses the volume name specified in the dump or load command. If neither **sp_volchanged** nor the command specifies a volume name, the Backup Server does not check this field in the ANSI tape label.

Volume Change Prompts for Dumps

This section describes the volume change prompts that occur while dumping a database or transaction log. For each prompt, it indicates the possible operator actions and the appropriate **sp_volchanged** response.

- `Mount the next volume to search.`

  When appending a dump to an existing volume, the Backup Server issues this message if it cannot find the end-of-file mark.

The operator can	By replying
Abort the dump	sp_volchanged *session_id*, *device_name*, abort
Mount a new volume and proceed with the dump	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Mount the next volume to write.`

  The Backup Server issues this message when it reaches the end of the tape. This occurs when it detects the end-of-tape mark, or dumps the number of kilobytes specified by the **capacity** parameter of the dump command, or dumps the *high* value specified for the device in the *sysdevices* system table.

The operator can	By replying
Abort the dump	sp_volchanged *session_id*, *device_name*, abort
Mount the next volume and proceed with the dump	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Volume on device` *`device_name`* `has restricted access (code` *`access_code`*`).`

  Dumps that specify the **init** option overwrite any existing contents of the tape. Backup Server issues this message if you try to dump to a tape with ANSI access restrictions without specifying the **init** option.

The operator can	By replying
Abort the dump	sp_volchanged *session_id*, *device_name*, abort
Mount another volume and retry the dump	sp_volchanged *session_id*, *device_name*, retry [, *file_name* [, *volume_name*]]
Proceed with the dump, overwriting any existing contents	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Volume on device` *`device_name`* `is expired and will be overwritten.`

  Dumps that specify the **init** option overwrite any existing contents of the tape. During dumps to single-file media, Backup Server issues this message if you have not specified the **init** option and the tape contains a dump whose expiration date has passed.

The operator can	By replying
Abort the dump	sp_volchanged *session_id*, *device_name*, abort

The operator can	By replying
Mount another volume and retry the dump	sp_volchanged *session_id*, *device_name*, retry [, *file_name* [, *volume_name*]]
Proceed with the dump, overwriting any existing contents	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Volume to be overwritten on '`*`device_name`*`' has not expired: creation date on this volume is `*`creation_date`*`, expiration date is `*`expiration_date`*`.`

  On single-file media, the Backup Server checks the expiration date of any existing dump unless you specify the **init** option. The Backup Server issues this message if the dump has not yet expired.

The operator can	By replying
Abort the dump	sp_volchanged *session_id*, *device_name*, abort
Mount another volume and retry the dump	sp_volchanged *session_id*, *device_name*, retry [, *file_name* [, *volume_name*]]
Proceed with the dump, overwriting any existing contents	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Volume to be overwritten on '`*`device_name`*`' has unrecognized label data.`

  Dumps that specify the **init** option overwrite any existing contents of the tape. Backup Server issues this message if you try to dump to a new tape or a tape with non-Sybase data without specifying the **init** option.

The operator can	By replying
Abort the dump	sp_volchanged *session_id*, *device_name*, abort
Mount another volume and retry the dump	sp_volchanged *session_id*, *device_name*, retry [, *file_name* [, *volume_name*]]
Proceed with the dump, overwriting any existing contents	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

Volume Change Prompts for Loads

Following are the volume change prompts and possible operator actions during loads:

- `Dumpfile '`*`file_name`*`' section `*`volume_name`*` found instead of '`*`file_name`*`' section `*`volume_name`*`.`

  The Backup Server issues this message if it cannot find the specified file on a single-file medium.

The operator can	By replying
Abort the load	sp_volchanged *session_id*, *device_name*, abort
Mount another volume and try to load it	sp_volchanged *session_id*, *device_name*, retry [, *file_name* [, *volume_name*]]
Load the file on the currently mounted volume, even though it is not the specified file (not recommended)	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Mount the next volume to read.`

  The Backup Server issues this message when it is ready to read the next section of the dump file from a multi-volume dump.

The operator can	By replying
Abort the load	sp_volchanged *session_id*, *device_name*, abort
Mount the next volume and proceed with the load	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

- `Mount the next volume to search.`

  The Backup Server issues this message if it cannot find the specified file on multi-file medium.

The operator can	By replying
Abort the load	sp_volchanged *session_id*, *device_name*, abort
Mount another volume and proceed with the load	sp_volchanged *session_id*, *device_name*, proceed [, *file_name* [, *volume_name*]]

Recovering a Database: Step-by-Step Instructions

The symptoms of media failure are as variable as the causes. If only a single block on the disk is bad, your database may appear to function perfectly for some time after the corruption appears, unless you're running **dbcc** commands frequently. If an entire disk or disk controller is bad, you will not be able to **use** a database. SQL Server marks it as suspect and displays a warning message. If the disk storing the *master* database fails, users will not be able to log into the server, and users already logged in will not be able to perform any actions that access the system tables in *master*.

This section describes what to do when a database device fails. The recommended procedure consists of the following steps:

1. Get a current log dump of **every database on the device**.
2. Examine the space usage of **every database on the device**.
3. Once you have gathered this information for all databases on the device, drop each database.
4. Drop the failed device.
5. Initialize new devices.
6. Recreate the databases, one at a time.
7. Lock users out of each database.
8. Load the most recent database dump into each database.
9. Apply each transaction log dump in the order it was created.

These steps are described in greater detail in the following sections.

Getting a Current Dump of the Transaction Log

Use **dump transaction with no_truncate** to get a current transaction log dump **for each database on the failed device**. For example, to get a current transaction log dump of *mydb*:

```
dump transaction mydb
   to "/dev/nrmt0" at REMOTE_BKP_SERVER
   with init, no_truncate,
   notify = "operator_console"
```

Examining the Device Allocations

The following steps are recommended, but not required, to determine which devices your database uses, how much space is allocated on each device, and whether the space is used for data, log, or both. You can use this information when recreating your databases, to ensure that your log, data, and indexes reside on separate devices.

1. In *master*, use the following query to examine the device allocations and uses for the damaged database:

   ```
   select segmap, size from sysusages
       where dbid = db_id("database_name")
   ```

2. Examine the output of the query. Each row with a *segmap* of 3 represents a data allocation; each row with a *segmap* of 4 in the represents a log allocation. (Higher values indicate user-defined segments; treat these as data allocations.) The *size* column indicates the number of 2K blocks of data. To find the number of megabytes, divide by 512 on most systems, or by 256 on Stratus. Note the order, use and size of each disk piece. For example, this output:

   ```
   segmap        size
   -------       --------
        3          10240
        3           5120
        4           5120
        3           1024
        4           2048
   ```

   translates into these sizes and uses, in megabytes:

Device Allocation	Megabytes
Data	20
Data	10
Log	10
Data	2
Log	4

Table 8-15: Sample Device Allocation

➤ ***Note***

If the *segmap* column contains 7s, your data and log are on the same device, and you can only recover up to the point of the most recent database dump. **Do not** use the **log on** option to **create database**. Just be sure that you allocate as much (or more) space than the total reported from *sysusages*.

3. Run **sp_helpdb** *database_name* for the database. This query lists the devices on which the data and logs are located:

```
name       db_size  owner    dbid    created
-------    -------  ------   ----    -----------
mydb       46.0 MB  sa         15    Apr  9 1991

status         device_fragments    size        usage
-------------- ----------------    ------      ------------
no options set datadev1            20 MB       data only
               datadev2            10 MB       data only
               datadev3            2 MB        data only
               logdev1             10 MB       log only
               logdev1             4 MB        log only
```

Dropping the Databases

Once you have performed these steps **for all databases on the failed device**, use the **drop database** command to drop each database.

➤ ***Note***

If tables in other databases contain references to any tables in the database you are trying to drop, you must remove the referential integrity constraints with ***alter table*** before you can drop the database.

If the system reports errors because the database is damaged when you issue the **drop database** command, use the **dropdb** option of the **dbcc dbrepair** command:

```
dbcc dbrepair (mydb, dropdb)
```

See the *Troubleshooting Guide* for more information on using **dbcc dbrepair.**

Dropping the Failed Devices

After you have dropped each database, use **sp_dropdevice** to drop the failed device. See Volume 2 of the *SQL Server Reference Manual* for information on this system procedure.

Initializing New Devices

Use **disk init** to initialize the needed new database devices. See Chapter 3, ‘‘Managing Physical Resources’’ for full information.

Recreating the Databases

Use the following steps to recreate each database using the segment information you collected earlier.

➤ ***Note***

If you chose not to gather information about segment usage, use **create database for load** to create a new database at least as large as the original.

1. Use the **create database** command with the **for load** option. Duplicate all segment mappings and sizes for each row of your database from the *sysusages* table, **up to and including the first log device**. Be sure to use the order of the rows as they appear in *sysusages*. (The results of **sp_helpdb** are in alphabetical order by device name, not in order of allocation.) For example, to recreate the *mydb* database:

   ```
   create database mydb
       on datadev1 = 20,
          datadev2 = 10
   log on logdev1 = 10
   for load
   ```

2. Use the **alter database** command with the **for load** option to recreate the remaining entries. In this example, to allocate more data space on *datadev3*, the command is:

   ```
   alter database on datadev3 = 2 for load
   ```

   To re-create the final log allocation on *logdev1*, use this command:

   ```
   alter database mydb log on logdev1 = 4 for load
   ```

Locking the Databases

If users perform logged transactions between loading the database and the first log dump, or between the loading of two transaction log dumps, the load fails. Lock each database before you load it to ensure that users do not make changes.

Use **sp_dboption** to set the **no chkpt on recovery, dbo use only,** and **read only** options to TRUE. Do not reset these options until the last transaction log has been loaded.

➤ ***Note***

create database... for load temporarily locks users out of the newly created database, but this protection is removed once **load database** completes.

Loading the Database

Reload the database using **load database.** If the original database stored objects on user-defined segments (*sysusages* reports a *segmap* > 4) and your new segment allocations do not match those in the dumped database, SQL Server remaps these segments as needed. This remapping may mix log and data on the same physical device, making recovery of the current log impossible.

If an additional failure occurs while a database is being loaded, SQL Server does not recover the partially loaded database, and notifies the user. You must restart the database load by repeating the **load** command.

Loading the Transaction Logs

Use **load transaction** to apply transaction log backups **in the same sequence in which they were made**. Load the most current dump last.

SQL Server checks the timestamps on each dumped database and transaction log. If the dumps are loaded in the wrong order, or if user transactions have modified the transaction log between loads, the load fails.

Once you have brought a database up to date, use **dbcc** commands to check its consistency.

Unlocking the Databases

After you have applied all transaction log dumps to a database, unlock it so that users can access it. Use **sp_dboption** to reset the **no chkpt on recovery, dbo use only** and **read only** options to FALSE.

9 Backing Up and Restoring the System Databases

Introdutio.n

This chapter is part of a three-chapter unit on backup and recovery. It explains how to restore the *master, model* and *sybsystemprocs* databases.

Depending on the database involved and the problems that you have on your system, recovery may include:

- Using **load database** to load backups of these databases.
- Using **buildmaster, installmaster** and **installmodel** to restore the initial state of these databases.
- A combination of both.

For more information about	See
Backup and recovery issues to address before production	Chapter 7, ‘‘Developing a Backup and Recovery Plan’’
dump, **load**, and **sp_volchanged** syntax	Chapter 8, ‘‘Backing Up and Restoring User Databases’’
Using thresholds to automate backups	Chapter 10, ‘‘Managing Free Space with Thresholds’’

Table 9-1: Further Information About Backup and Recovery

Symptoms of a Damaged *master* Database

A damaged *master* database can be caused by a media failure in the area on which *master* is stored, or by internal corruption in the database. A damaged *master* database makes itself known in one or more of these ways:

- SQL Server cannot start.
- There are frequent or debilitating segmentation faults or input/output errors.
- **dbcc** (the database consistency checker) reports damage during a regularly-scheduled check of your databases.

Avoiding Volume Changes During Backup and Recovery

When you dump the *master* database, be sure that the entire dump fits on a single volume, unless you have more than one SQL Server able to communicate with your Backup Server. You must start SQL Server in single-user mode before loading the *master* database. This does not allow a separate user connection to respond to Backup Server's volume change messages during the load. Since *master* is usually small in size, placing its backup on a single tape volume is typically not a problem.

Recovering the *master* Database

This section describes the step to recover the *master* database in two situations:

- When the database is corrupt but the master device has not been damaged. This process does not affect the *model* database. Recovering *model* is covered later in this chapter.
- When the master device is damaged, and you must restore the entire device. You can also use these procedures to move your master database to a larger master device.

Special procedures are needed because of the central, controlling nature of the *master* database and the master device. Tables in *master* configure and control all of SQL Server's functions, databases, and data devices. The recovery process:

- Restores *master* to the default state on a newly-installed server.
- Restores *master* to its condition at the time of your last backup.
- Performs very special procedures to recover changes to devices and databases since the last backup.

During the early stages of recovering the *master* database, you will not be able to use the system stored procedures.

If you have user databases that you have not backed up on your *master* device, you may not be able to use these procedures to recover. You should call Technical Support for assistance.

Summary of Recovery Procedure

System Administrators must use the following steps to restore a damaged *master* database. **Each of these steps is discussed in more detail in the following pages.**

1. Make hard copies of vital system tables needed to restore disks, databases and logins.
2. If there are other databases on the master device, and they are accessible, back them up with **dump database**.
3. Use **buildmaster** to build a new *master* database or master device.
4. Restart SQL Server in single-user mode with the **startserver** command.
5. If your *master* database is larger than 3MB, re-create its allocations in *sysusages* **exactly**. The size of *master* might be larger because of **alter database** commands, or because earlier upgrades of SQL Server required a larger *master* database.
6. If your Backup Server's network name (the name in the interfaces file) is **not** SYB_BACKUP, change the network name in *sysservers*.
7. Check to see that your Backup Server is running.
8. Use **load database** to load the most recent database dump of *master*. SQL Server stops automatically after successfully loading *master*.
9. Restart SQL Server in single-user mode again with **startserver**.
10. If you have added database devices since the last dump of *master*, issue the **disk reinit** command to rebuild *sysdevices*.
11. If you have run **disk reinit**, or if **create database** or **alter database** has been used since the last dump, make hard copies of *sysusages* and *sysdatabases* and then issue the **disk refit** command to rebuild these system tables.
12. Check for consistency: compare your hard copy of *sysusages* and *sysdatabases* with the new on-line version, run **dbcc checkalloc** on each database, and examine important tables in each database.
13. If you restored the entire master device, restore the *model* database.
14. Reload any affected user databases.
15. Check *syslogins* if you have added new logins since the last backup of *master*.

16. If everything is correct, stop the server and use **startserver** to restart SQL Server for multi-user use.
17. Dump the *master* database.

The following sections describe these steps in greater detail.

Saving Copies of System Tables

If you can log into your server, save copies of the following system tables to a disk file: *sysdatabases*, *sysdevices*, *sysusages*, *sysloginroles* and *syslogins*. You can use these to guarantee that your system has been fully restored at the completion of this process.

You can use **isql** (use **select * from** *tablename*) or **bcp** (copy the tables in character mode) to make copies to disk files. If possible, make a copy of *sysusages* ordered by *vstart*:

```
select * from sysusages order by vstart
```

For more information on the **isql** and **bcp** programs, see the *SQL Server Utility Programs* manual for your operating system.

Dumping User Databases on the Master Device

If there are user databases on the master device, and they are accessible, use **dump database** to dump them.

Building a New *master* Database

If you are rebuilding only your *master* database, you use a special option to **buildmaster** that only affects *master*. If you are rebuilding the entire master device, you use **buildmaster** without this option.

Rebuilding Only the master Database

Run **buildmaster -m** (UNIX and PCs) or **buildmaster /master** (OpenVMS) to replace the damaged *master* database with a copy of a “generic” *master* database. Give the full name for your master device, and the full size of the device.

➤ ***Note***

You must give buildmaster a size at least as large, or larger, then the size originally used to configure SQL Server. If you recorded this information in your *SQL Server Installation Guide*, use that size. If the size you provide is too small, you will get error messages when you try to load your databases.

This example rebuilds the *master* database on a 17MB (8704 2-K pages) master device:

On UNIX platforms:

```
buildmaster -d /dev/rsd1f -s8704 -m
```

On OpenVMS:

```
buildmaster
/disk=dua0:[devices.master]d_master.dat/size=8704
/master
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

After you run **buildmaster**, the password for the default “sa” account reverts to NULL.

For details on the **buildmaster** utility, see the *SQL Server Utility Programs* manual for your operating system.

Rebuilding the Entire Master Device

If you are rebuilding a damaged master device, or moving your master database to another device, run **buildmaster** without using the “master” option.

Starting SQL Server in Master-Recover Mode

Start SQL Server in master-recover mode with the **-m** (UNIX and PCs) or **/masterrecover** (OpenVMS) options. See the *SQL Server Utility Programs* manual for your operating system for complete syntax.

On UNIX platforms:

```
startserver -f RUN_server_name -m
```

On OpenVMS:

```
startserver /server = server_name /masterrecover
```

When you start SQL Server in master-recover mode, only one login of one user—the System Administrator—is allowed. Immediately after a **buildmaster** command on the *master* database, only the "sa" account exists, and its password is NULL.

This special master-recover mode is necessary because the generic *master* database created with **buildmaster** does not match the actual situation on SQL Server: it doesn't know about any of your database devices, for example! Any operations on the *master* database could make recovery impossible, or at least much more complicated and time-consuming.

A SQL Server started in master-recover mode is automatically configured to allow direct updates to the system tables. Certain other operations (for example, the checkpoint process) are disallowed.

◆ ***WARNING!***

Ad hoc changes to the system tables are dangerous—some changes can render SQL Server unable to run. Make only the changes described in this chapter, and always make the changes in a user-defined transaction.

Re-creating Devices Allocations for *master*

Look at a hard copy of *sysusages* created before you ran **buildmaster**, or your most recent copy of this table. If it has only one line for *dbid* 1, your *master* database size has not been changed, and you can skip to the next step, ''Checking Your Backup Server sysservers Information'' on page 9-11.

◆ ***WARNING!***

If you have increased the size of *master*, and have user databases on the master device that are not backed up, you have a difficult recovery situation. You should call Technical Support for assistance before you proceed.

If you have more than one row for *dbid* 1 in your hardcopy of *sysusages*, you need to increase the size of *master* so that you can load the dump. You must duplicate the *vstart* value for each allocation for *master* in *sysusages*. This is easiest to do if you have a copy of *sysusages* ordered by *vstart*.

In the simplest cases, additional allocations to *master* only require the use of **alter database.** In more complicated situations, you must allocate space for other databases in order to reconstruct the exact *vstart* values needed to recover *master.*

In addition to the *master* database, *tempdb* (*dbid* =2) and *model* (*dbid* = 3) are located wholly or completely on the master device. In addition, there may be user databases wholly or partially on the device.

Determining Which Allocations Are on the Master Device

To determine which *vstart* values represent allocations on the master device, look at your hardcopy of the *sysdevices* table. It shows the low and high values for each device (the rows for tape devices aren't included in this sample):

```
low         high        status cntrltype name
        phyname
        mirrorname
----------- ----------- ------ --------- -------
       ----------------------------------------
       ----------------------------------------
          0        8703      3         0 master
        d_master
        NULL
   16777216    16782335      2          0 sprocdev
        /sybase/new10_sprocdev
        NULL
```

Page Range for Master Device

Page numbers on the master device are between 0 and 8703, so any database allocation with *sysusages.vstart* values in this range represent allocations on the master device.

A Simple Case: Only master Altered

Check all rows for *master* (except the first) in your saved *sysusages* output. Here is sample *sysusages* information, ordered by *vstart*:

```
dbid segmap lstart size   vstart     pad  unreservedpgs
---- ------ ------ ------ ---------- ---- -------------
   1      7      0   1536          4 NULL           480
   3      7      0   1024       1540 NULL           680
   2      7      0   1024       2564 NULL           680
   1      7   1536   1024       3588 NULL          1024
   4      7      0   5120   33554432 NULL          1560
```

dbid = 1, an additional allocation for master

size of this allocation

vstart between 0 and 8703

In this simple example, the first four rows have *vstart* values between 0 and 8703. Only *dbid* 4 is on another device.

buildmaster re-creates the first three rows, so *sysusages* in your newly-rebuilt *master* database should match your hardcopy.

The fourth row shows an additional allocation for *master*, with *vstart* = 3588 and *size* = 1024.

Figure 9-1 shows the storage allocations for the above *sysusages* data.

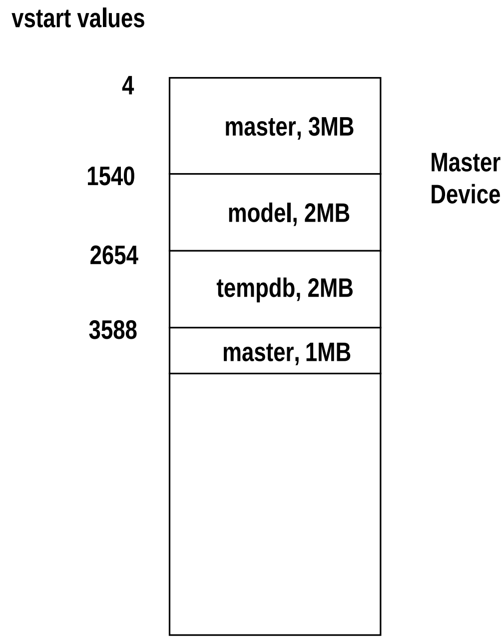


Figure 9-1: Allocations on a Master Device

In this simple case, you only need to issue an **alter database** command to increase the size of the *master* database. To determine the size to provide for this command, look at the *size* column for the second allocation to *master*. Divide by 512 (use 256 on Stratus). In this example, the additional row for *master* indicates an allocation of 1024 data pages, so the correct parameter is 2, the result of 1024/512.

Use that result for the **alter database** command. Log on to the server as "sa". Remember that **buildmaster** set the password for this account to NULL. Issue the **alter database** command. For the example above, use:

```
alter database on master = 2
```

Check the *size* and *vstart* values for the new row in *sysusages*.

A More Complex Allocation

Your output from *sysusages* will have more allocations on the master device if:

- Your SQL Server has been upgraded from earlier SQL Server releases
- A System Administrator has increased the size of *master, model* and / or *tempdb* on the master device.
- A user database has been created or altered on the master device.

You must restore these allocations up to the last row for the *master* database, *dbid* 1. Here is an example of *sysusages* showing additional allocations on the master device, in order by *vstart*:

```
dbid segmap lstart size  vstart   pad  unreservedpgs
---- ------ ------ ----- -------- ---- -------------
   1      7      0  1536        4 NULL           472
   3      7      0  1024     1540 NULL           680
   2      7      0  1024     2564 NULL           680
   1      7   1536   512     3588 NULL           512
   5      7      0  1024     4100 NULL           680
   2      7   1024  1024     5124 NULL          1024
   1      7   2048   512     6148 NULL           512
   4      7      0  5120 16777216 NULL          1560
```

dbid = 1, additional allocations for master

vstart between 0 and 8703

This copy of *sysusages* shows the following allocations on the master device, (excluding the three created by **buildmaster**):

- One for *master, dbid* = 1, *size* = 512, *vstart* = 3588
- One for a user database, *dbid* = 5, *size* = 1024, *vstart* = 4100 (consult your hardcopy *sysdatabases* to find the name of this database)
- One for *tempdb, dbid* = 2, *size* = 1024, *vstart* = 5124
- Another allocation for *master, dbid* = 1, *size* = 512, *vstart* = 6148

The final allocation in this output is not on the master device.

Figure 9-2 shows the allocations on the master device.

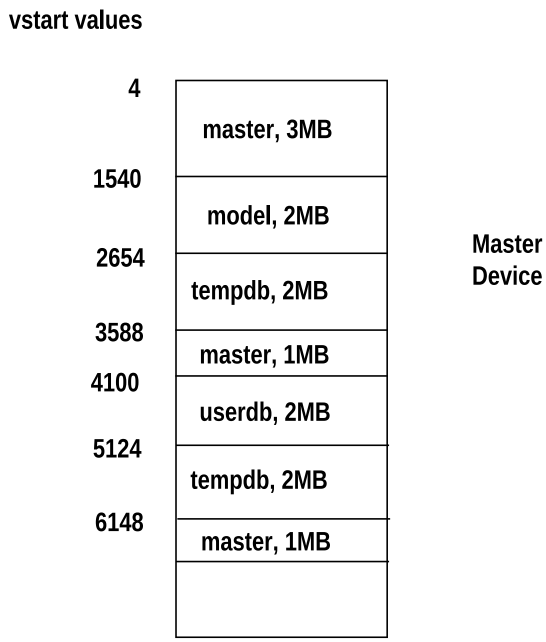


Figure 9-2: Complex Allocations on a Master Device

You need to issue a set of **alter database** and **create database** commands to recreate all of the allocations. If your *sysusages* table lists additional allocations on the master device after the last allocation for master, note that you do not have to recreate them.

To determine the size for the **create database** and **alter database** commands, divide the value shown in the *size* column of the *sysusages* output by 512 (256 on Stratus). To reconstruct the allocation, issue these commands, in this order:

To restore the first allocation to *master, dbid* 1, *size* = 512:

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

To create a user database, *dbid* 5, *size* = 1024:

```
create database userdb on default = 2
```

To allocate more space to *tempdb, dbid* 2, *size* = 1024:

```
alter database tempdb on default = 2
```

To add the final allocation to *master, dbid* 1, *size* = 512:

```
alter databse master on default = 1
```

You only need to restore all allocations up to and including the last line for the *master* device. When you load the backup of *master*, this table is completely restored from the dump.

◆ ***WARNING!***

This set of steps will wipe out the data in the user database. on the master device. If it has not been backed up since it was last changed, do not proceed. Technical Support may be able to recover for you.

At this point, carefully check the current *sysusages* values with the values in your hardcopy:

- If all of the *vstart* values for *master* match, proceed to the next step.
- If the values do not match, an attempt to load the *master* database will almost certainly fail. Shut down the server, and begin again by running **buildmaster**. See ‘‘Building a New master Database’’ on page 9-4.

If your *sysusages* values look correct, proceed to the next step.

Checking Your Backup Server *sysservers* Information

Log on to the server as “sa”, using the NULL password.

If the network name of your Backup Server is not SYB_BACKUP, you must update *sysservers* so that your SQL Server can communicate with its Backup Server. Check the Backup Server name in your interfaces file, and issue this command on SQL Server:

```
select *
from sysservers
where srvname = "SYB_BACKUP"
```

Check the *srvnetname* in the output from this command. If it matches the interfaces file entry for the Backup Server for your server, skip to the next step.

If the *srvnetname* reported by this command is **not** the same as the name of your Backup Server in the interfaces file, you must update *sysservers*.The example below changes the Backup Server’s network name to PRODUCTION_BSRV:

```
begin transaction

update sysservers
set srvnetname = "PRODUCTION_BSRV"
where srvname = "SYB_BACKUP"
```

Execute this command, and check to see that it modified only one row. Issue the **select** command again, and check to be sure that the correct row was modified, and that it contains the correct value. If the

update command modified more than one row, or if it modified the wrong row, issue a **rollback transaction** command, and attempt the update again.

If the command correctly modified the Backup Server's row, issue a **commit transaction** command.

Checking for a Running Backup Server

Use the **showserver** command from your operating system to verify that your Backup Server is running, and restart your Backup Server if necessary. See **showserver** and **startserver** in the *SQL Server Utility Programs* manual for your operating system.

Loading a Backup of *master*

Load the most recent backup of the *master* database with **load database**. Here are examples of the load commands:

On UNIX platforms:

```
load database master from "/dev/nrmt4"
```

On OpenVMS:

```
load database master from "MTA0:"
```

See Chapter 8, "Backing Up and Restoring User Databases" for information on command syntax. See the *System Administration Guide Supplement* for PC platform examples.

After the **load database** command completes successfully, SQL Server automatically shuts itself down. Watch for any error messages during the load, and during the shutdown.

Restarting SQL Server in Master-Recover Mode

Use **startserver** to restart SQL Server in master-recover mode. Watch for error messages during recovery.

Check the *sysusages*, *sysdatabases* and *sysdevices* tables in your recovered server against your hardcopy. Look especially for these problems:

- If there are devices in your hardcopy that are not included in the restored *sysdevices*, then you have added devices since your last backup, and you must run **disk reinit** and **disk refit**.

- If there are databases listed in your hardcopy that are not listed in your restored *sysdatabases* table, you have added a database since the last time you backed up *master*. You must run **disk refit**.

Loading the backup of *master* restores the “sa” account to its previous state. It restores the password on the “sa” account, if any. If you used **sp_locklogin** to lock this account before the backup was made, the “sa” account will now be locked. Perform the rest of these steps using an account with the System Administrator role.

Re-Adding Database Devices

If you have added any database devices since the last dump—that is, if you have issued a **disk init** command—you must add each new device to *sysdevices* with the **disk reinit** command. If you save scripts from your original **disk init** commands, use them to determine the parameters for the **disk reinit** command. If the size you provide is too small, you may corrupt your database.

If you do not save your **disk init** scripts, look at your most recent hardcopy of *sysdevices* to determine the correct parameters for **disk reinit**.

disk reinit parameter	*sysdevices data*	Notes
name	*name*	Use the same name, especially if you have any scripts which create or alter databases, or add segments.
physname	*phyname*	Must be full path to device.
vdevno	*low* / 16777216	Not necessary to use the same value for *vdevno*, but you must be sure to use a value not already in use.
size	(*high-low*) +1	Extremely important to provide correct size information.

Table 9-2: Using sysdevices to Determine disk reinit Parameters

If you store your *sybsystemprocs* database on a separate physical device, be sure to include a **disk reinit** command for *sybsystemprocs* if it is not listed in *sysdevices* at this point.

After running **disk reinit**, compare your *sysdevices* table to the copy you made before running **buildmaster**.

disk reinit can only be run from the *master* database, and only by a System Administrator. Permission cannot be transferred to other users. Its syntax is:

```
disk reinit
     name = "device_name",
     physname = "physical_name",
     vdevno = virtual_device_number,
     size = number_of_blocks
     [, vstart = virtual_address,
     cntrltype = controller_number]
```

For more on **disk reinit**, see the discussion of **disk init** in Chapter 3, “Managing Physical Resources”, or Volume 1 of the *SQL Server Reference Manual*.

Rebuilding *sysusages* and *sysdatabases*

If you have added database devices or created or altered databases since the last database dump, use **disk refit** to rebuild the *sysusages* and *sysdatabases* tables.

disk refit can be run only from the *master* database and only by a System Administrator. Permission cannot be transferred to other users. Its syntax is:

```
disk refit
```

◆ ***WARNING!***

Providing inaccurate information in the disk reinit command may lead to permanent corruption when you update your data! Be sure to check your SQL Server with dbcc after running disk refit.

Checking SQL Server

Check SQL Server carefully:

1. Compare your hard copy of *sysusages* with the new on-line version.
2. Compare your hard copy of *sysdatabases* with the new on-line version.
3. Run **dbcc checkalloc** on each database.
4. Examine important tables in each database.

◆ ***WARNING!***

If you find discrepancies in *sysusages*, call Technical Support for help.

If you find other problems, rerun **disk reinit** and **disk refit** and check your SQL Server again.

Restoring *model*

If you are restoring only the *master* database, you can skip this step.

If you are restoring the entire master device, you must also restore *model*:

- Load your backup of *model*, if you keep a backup.
- If you do not have a backup:
  - Run the **installmodel** script:

    On UNIX platforms:

    ```
    cd $SYBASE/scripts
    setenv DSQUERY server_name
    isql -Usa -Ppassword -Sserver_name < installmodel
    ```

    On OpenVMS:

    ```
    set default sybase_system:[sybase.scripts]
    define dsquery server_name
    isql/user="sa"/password="password"
     /input=installmodel
    ```

  - Re-do any changes you have made to *model*.

For PC platform examples, refer to the *System Administration Guide Supplement*.

Loading User Databases

If you had user databases stored on the master device, and had to issue **create database** commands during the previous steps, reload your user databases with your usual load commands.

Restoring Server User IDs

Check your hardcopy of *syslogins* and your restored *syslogins* table. Look especially for the following:

- If you have added server logins since the last backup of *master*, reissue the **sp_addlogin** commands.
- If you have dropped server logins, reissue the **sp_droplogin** commands.
- If you have locked server accounts, reissue the **sp_locklogin** commands.
- There may be other differences due to use of the **sp_modifylogin** procedure, by users or by System Administrators.

It is very important to ensure that users receive the same *suid*. Mismatched *suid* values in databases can lead to permission problems, and users may not be able to access tables or run commands.

An effective technique for checking existing *suid* values is to perform a **union** on the *sysusers* tables in your user databases. You can include *master* in this procedure, if users have permission to use *master*.

For example:

```
select suid, name from master..sysusers
union
select suid, name from sales..sysusers
union
select suid, name from parts..sysusers
union
select suid, name from accounting..sysusers
```

If your resulting list shows skipped *suid* values in the range where you need to redo the logins, you must add placeholders for the skipped values, and then drop them with **sp_droplogin**, or lock them with **sp_locklogin.**

Restarting SQL Server

Once you have finished restoring the *master* database, use the **shutdown** command to shut down SQL Server. Then use **startserver** to restart in multi-user mode.

Backing Up *master*

When you have completely restored the *master* database and have run full **dbcc** integrity checks, back up the database using your usual dump commands.

Recovering the *model* Database

This section describes recovery of the *model* database when only the *model* database needed to be restored. It includes instructions for these scenarios:

- You have not made any changes to *model*, so you only need to restore the generic *model* database.
- You have changed *model*, and have a backup.
- You have changed *model*, and do not have a backup.

Restoring the Generic *model* Database

buildmaster can restore the *model* database without affecting *master*. You must always shut down the server before using **buildmaster.**

➤ ***Note***

Be sure to shut down SQL Server before running buildmaster.

On UNIX platforms:

```
buildmaster -d /devname -x
```

On OpenVMS:

```
buildmaster /disk = physicalname /model
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

Restoring *model* from a Backup

If you can issue the **use model** command successfully, you can restore your *model* database from a backup with the **load database** command.

If you cannot use the database:

1. Follow the instructions above for using **buildmaster** to restore the *model* database.
2. If you have changed the size of model, re-issue the **alter database** commands.
3. Load the backup with **load database**.

Restoring *model* with No Backup

If you have changed your *model* database and do not have a backup, you should:

- Follow the steps above for restoring a generic model database.
- Re-issue all of the commands you issued to change *model*.

Recovering the *sybsystemprocs* Database

The *sybsystemprocs* database stores the system procedures that are used to modify and report on system tables. If your routine **dbcc** checks on this database report damage, and you do not keep a backup of this database, you can restore it using **installmaster**. If you do keep backups of *sybsystemprocs*, you can restore it with **load database.**

Restoring *sybsystemprocs* with *installmaster*

1. Check to see what logical device currently stores the database. If you can still use **sp_helpdb**, issue this command:

   `sp_helpdb sybsystemprocs`

```
name                db_size       owner            dbid
        created
        status
------------------- ------------- ---------------- ------
sybsystemprocs            10.0 MB sa                    4
        Aug 07, 1993
        trunc log on chkpt

device_fragments   size        usage            free kbytes
------------------ ----------- ---------------- -----------
sprocdev           10.0 MB     data and log            3120
```

   The *device_fragments* column indicates that the database is stored on *sprocdev*.

   If you cannot use **sp_helpdb**, this query reports the devices used by the database, and the amount of space on each device:

```
select sysdevices.name, sysusages.size / 512
from sysdevices, sysdatabases, sysusages
where sysdatabases.name = "sybsystemprocs"
  and sysdatabases.dbid = sysusages.dbid
  and sysdevices.low <= sysusages.size + vstart
  and sysdevices.high >= sysusages.size + vstart -1
```

```
name
--------------- -------
sprocdev              10
```

On Stratus, use "256" instead of "512" on the first line in the query above.

2. Drop the database:

```
drop database sybsystemprocs
```

If the physical disk is damaged, use **sp_dropdevice** to drop the device. If necessary, use **disk init** to initialize a new database device. See Chapter 3, ''Managing Physical Resources'', for more information on **disk init**

3. Re-create the *sybsystemprocs* database on the device, using the size returned by the query above:

```
create database sybsystemprocs
    on sprocdev = 10
```

4. Run the **installmaster** script.

On UNIX platforms:

```
cd $SYBASE/scripts
setenv DSQUERY server_name
isql -Usa -Ppassword -Sserver_name < installmaster
```

On OpenVMS:

```
set default sybase_system:[sybase.scripts]
define dsquery server_name
isql/user="sa"/password="password"
    /input=installmaster
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

5. If you have made any changes to permissions in *sybsystemprocs*, or if you have added your own procedures to the database, you must re-do the changes.

Restoring *sybsystemprocs* with *load database*

If you write system procedures and store them in the *sybsystemprocs* database, you have two ways to recover them if the database is damaged:

- Restore the database from **installmaster**, as described above, and then re-create the procedures by re-issuing the **create procedure** commands.

- Keep backups of the database, and load them with **load database**

If you choose to keep a backup of the database, you should insure that the complete backup fits on one tape volume, or that you have more than one SQL Server able to communicate with your Backup Server. If a dump spans more than one tape volume, you issue the change-of-volume command using system procedure, **sp_volchanged**, which is stored in *sybsystemprocs*. You can't issue that command in the middle of recovering this database.

Here are sample load commands:

On UNIX:

```
load database sybsystemprocs from "/dev/nrmt4"
```

On OpenVMS:

```
load database sybsytemprocs from "MTA0:"
```

For PC platform examples, refer to the *System Administration Guide Supplement*.

10 Managing Free Space with Thresholds

Introduction

When you create or alter a database, you allocate a finite amount of space for its data and log segments. As you create objects and insert data, the amount of free space in the database decreases.

This chapter is part of a four-chapter unit on backup and recovery. It explains how to use thresholds to monitor the amount of free space in a database segment. It describes the **last-chance threshold**, which helps ensure that you have enough space to dump the transaction log, and explains how to use additional thresholds to detect space shortages. It also provides guidelines for creating threshold procedures.

For more information about	See
Backup and recovery issues to address before production	Chapter 7, "Developing a Backup and Recovery Plan"
dump, **load**, and **sp_volchanged** syntax	Chapter 8, "Backing Up and Restoring User Databases"
Backing up and restoring the system databases	Chapter 9, "Backing Up and Restoring the System Databases"

Table 10-1: Further Information About Backup and Recovery

Monitoring Free Space with the Last-Chance Threshold

A threshold always has a stored procedure associated with it, and the threshold acts like a trip-wire. When free space on the segment falls below the amount specified by the threshold, SQL Server executes the stored procedure.

Each database that stores its transaction log on a separate segment has a last-chance threshold. The threshold is an estimate of the number of free log pages that would be required to back up the transaction log. SQL Server automatically adjusts the last-chance threshold as you allocate more space to the log segment.

When the amount of free space in the log segment falls below the last-chance threshold, SQL Server automatically executes a special stored procedure called **sp_thresholdaction**. (You can specify a different last-chance threshold procedure with **sp_modifythreshold**.)

Figure 10-1 illustrates a log segment with a last-chance threshold. The shaded area represents log space that has already been used; the unshaded area, free log space. The last-chance threshold has not yet been crossed:

Figure 10-1: Log Segment with a Last-Chance Threshold

Crossing the Threshold

As users execute transactions, the amount of free log space decreases. When the amount of free space crosses the last-chance threshold, SQL Server automatically executes **sp_thresholdactio**n:

Figure 10-2: Executing sp_thresholdaction when the Last-chance Threshold is Reached

Controlling How Often *sp_thresholdaction* Executes

SQL Server uses a “hysteresis value”, the global variable *@@thresh_hysteresis*, to control how sensitive thresholds are to variations in free space. Once a threshold executes its procedure, it is deactivated. The threshold remains inactive until the amount of free space in the segment rises *@@thresh_hysteresis* pages above the threshold. This prevents thresholds from executing their procedures repeatedly in response to minor fluctuations in free space.

When the threshold in Figure 10-2 executes **sp_thresholdaction**, it is deactivated. In Figure 10-3, the threshold is reactivated when the amount of free space increases by @@*thresh_hysteresis* pages:

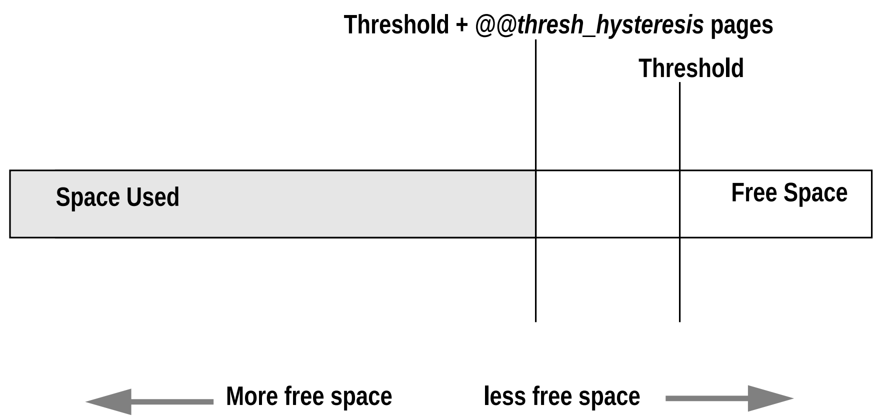


Figure 10-3: Free Space must Rise by @@thresh_hysteresis to Reactivate Threshold

Choosing Whether to Abort or Suspend Processes

By design, the last-chance threshold allows enough free log space to record a **dump transaction** command. There may not be enough room to record additional user transactions against the database.

When the last-chance threshold is crossed, SQL Server suspends user processes and displays the message:

```
Space available in the log segment has fallen
critically low in database 'mydb'. All future
modifications to this database will be suspended
until the log is successfully dumped and space
becomes available.
```

Only commands that are not recorded in the transaction log (**select**, fast **bcp**, **readtext**, and **writetext)** and commands that might be necessary to free additional log space (**dump transaction**, **dump database**, and **alter database**) can be executed.

Aborting Processes

To abort user processes rather than suspending them, use the **abort tran on log full** option of **sp_dboption** followed by the **checkpoint** command. For example, to abort processes in *mydb* when the last-chance threshold is crossed:

```
sp_dboption mydb, "abort tran on log full", true
use mydb
checkpoint
```

Waking Suspended Processes

Once the **dump transaction** command frees sufficient log space, suspended processes automatically awaken and complete. If fast **bcp**, **writetext**, or **select into** has resulted in unlogged changes to the database since the last backup, the last-chance threshold procedure cannot execute a **dump transaction** command. When this occurs, make a copy of the database with **dump database**, then truncate the log with **dump transaction.**

If this does not free enough space to awaken the suspended processes, it may be necessary to increase the size of the transaction log. Use the **log on** option of the **alter database** command to allocate additional log space.

As a last resort, System Administrators can use the **sp_who** command to determine which processes are suspended and the following statement to awaken sleeping processes:

```
select lct_admin("unsuspend", db_id)
```

◆ ***WARNING!***

Use the lct_admin function with extreme caution.

After you issue this command, the transactions continue, and may completely fill up the transaction log. In order to kill suspended processes, first issue the **kill** command, and then execute the command above. See "Killing Processes" on page 11-12 for more information.

Adding, Changing, and Deleting Thresholds

The Database Owner or System Administrator can create additional thresholds to monitor free space on any segment in the database. Each database can have up to 256 thresholds, including the last-chance threshold.

The **sp_addthreshold**, **sp_modifythreshold**, and **sp_dropthreshold** system procedures allow you to create, change, and delete thresholds. To prevent users from accidentally affecting thresholds in the wrong database, these procedures all require you to specify the name of the current database.

Displaying Information About Existing Thresholds

Use the system procedure **sp_helpthreshold** for information about all thresholds in a database. Use **sp_helpthreshold** *segment_name* for information about the thresholds on a particular segment.

The following example displays information about the thresholds on the database's default segment. Since "default" is a reserved word, it must be enclosed in quotation marks. The output of shows that there is one threshold on this segment set at 200 pages. The zero in the *last chance* column indicates that this is not a last-chance threshold:

```
sp_helpthreshold "default"

name        free_space  last chance?  proc_name
-----       ----------  -----------   ---------
default     200         0            space_dataseg

(1 row affected)
```

Adding a Threshold

Use the system procedure **sp_addthreshold** to create new thresholds. Its syntax is:

sp_addthreshold ***database_name, segment_name, free_pages, proc_name***

The *database_name* must specify the name of the current database. The remaining parameters specify the segment whose free space is being monitored, the size of the threshold in database pages, and the name of a stored procedure.

When the amount of free space on the segment falls below the threshold, an internal SQL Server process executes the associated procedure. This process has the permissions of the user who created the threshold, at the time he or she executed **sp_addthreshold**, less any permissions that have since been revoked.

Thresholds can execute a procedure in the same database, in another user database, in *sybsystemprocs* or in *master*. They can also call a remote procedure on an Open Server. **sp_addthreshold** does not verify that the threshold procedure exists at the time you create the threshold.

Changing a Threshold

Use the system procedure **sp_modifythreshold** to associate a threshold with a new threshold procedure, free-space value, or segment. **sp_modifythreshold** drops the existing threshold and creates a new one in its place. Its syntax is:

```
sp_modifythreshold database_name, segment_name,
   free_pages [, new_procedure [, new_free_pages
   [, new_segment]]]
```

database_name must be the name of the current database. *segment_name*, and *free_pages* identify which threshold you want to change.

For example, to execute a threshold procedure when free space on the segment falls below 175 pages rather than below 200 pages:

```
sp_modifythreshold mydb, "default", 200, NULL, 175
```

In this example, NULL acts as a placeholder so that *new_free_pages* falls in the correct place in the parameter list. The name of the threshold procedure is not changed.

The person who modifies the threshold becomes the new threshold owner. When the amount of free space on the segment falls below the threshold, SQL Server executes the threshold procedure with the owner's permissions at the time he or she executed **sp_modifythreshold**, less any permissions that have since been revoked.

Specifying a New Last-Chance Threshold Procedure

You can use **sp_modifythreshold** to change the name of the procedure associated with the last-chance threshold. You **cannot** use it to change the amount of free space or the segment name for the last-chance threshold.

sp_modifythreshold requires that you specify the number of free pages associated with the last-chance threshold. Use **sp_helpthreshold** to determine this value.

The following example displays information about the last-chance threshold, then specifies a new procedure, **sp_new_thresh_proc**, to execute when the threshold is crossed:

```
sp_helpthreshold logsegment
segment name   free pages   last chance?   threshold procedure
------------   ----------   ------------   -------------------
logsegment      40                    1   sp_thresholdaction
(1 row affected, return status = 0)
sp_modifythreshold mydb, logsegment, 40,
sp_new_thresh_proc
```

Dropping a Threshold

Use the system procedure **sp_dropthreshold** to remove a free-space threshold from a segment. Its syntax is:

sp_dropthreshold *database_name, segment_name, free_pages*

The *database_name* must specify the name of the current database. You must specify both the segment name and the number of free pages, since there can be several thresholds on a particular segment. For example:

```
sp_dropthreshold mydb, "default", 200
```

Creating an Additional Threshold for the Log Segment

When the last-chance threshold is crossed, all transactions are aborted or suspended until sufficient log space is freed. In a production environment, this can have a heavy impact on users. Adding a second, correctly placed threshold on your log segment can minimize the chances of crossing the last-chance threshold (and of blocking user transactions).

The additional threshold should dump the transaction log often enough so that the last-chance threshold is rarely crossed. It should not dump it so often that restoring the database requires the loading of too many tapes.

This section helps you determine the best place for a second log threshold. It starts by adding a threshold set at 50% of log capacity, and adjusts this threshold based upon space usage at your site.

Adding a Log Threshold at 50% of Available Space

Use the following procedure to add a log threshold at 50% of capacity:

1. Use the following query to determine the log's capacity in pages:

```
select sum(size)
from master..sysusages
where dbid = db_id("database_name")
and (segmap & 4) = 4
```

2. Use **sp_addthreshold** to add a new threshold at 50% of log capacity. For example, if the log's capacity is 2048 pages, add a threshold at 1024 pages:

```
sp_addthreshold mydb, logsegment, 1024, thresh_proc
```

3. Use the **create procedure** statement to create a simple threshold procedure that dumps the transaction log to the appropriate devices. (For more information about creating threshold procedures, see the section ‘‘Creating Threshold Procedures’’ later in this chapter.)

Testing and Adjusting the New Threshold

Use the **dump transaction** command to make sure your transaction log is less than 50% full. Then use the following procedure to test the new threshold:

1. Fill the transaction log by simulating routine user action. Use automated scripts that perform typical transactions at the projected rate.

   When the 50% threshold is crossed, your threshold procedure will dump the transaction log. Since this is not a last-chance threshold, transactions will not be suspended or aborted; the log will continue to grow during the dump.

2. While the dump is in progress, use **sp_helpsegment** to monitor space usage on the log segment. Record the maximum size of the transaction log just before the dump completes.

3. If there was considerable space left in the log when the dump completed, you may not need to dump the transaction log so soon:

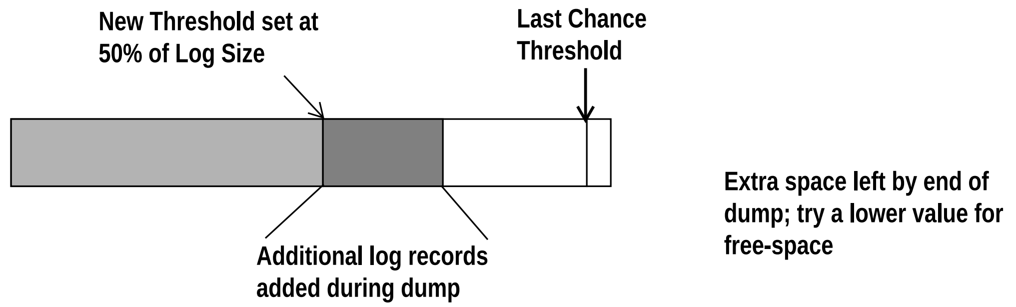


Figure 10-4: Transaction Log with Additional Threshold at 50%

Try waiting until only 25% of log space remains:

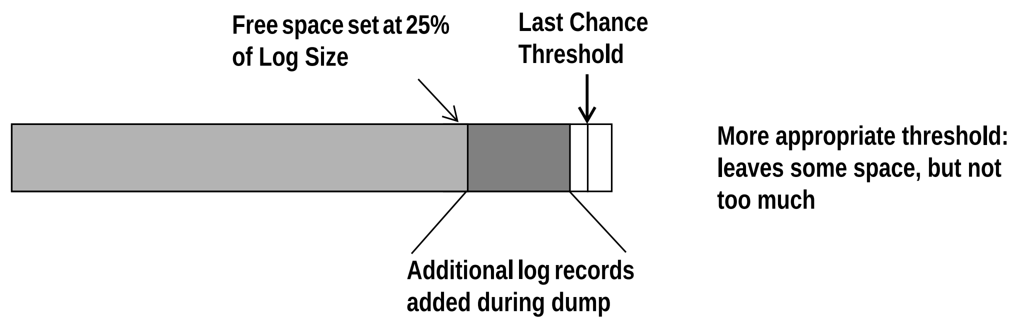


Figure 10-5: Moving Threshold Leaves Less Free Space after Dump

Use **sp_modifythreshold** to adjust the free space value to 25% of log capacity. For example:

```
sp_modifythreshold mydb, logsegment, 512,
    thresh_proc
```

4. Dump the transaction log and test the new free space value. If the last chance threshold is crossed before the dump completes, you are not beginning the **dump transaction** soon enough:

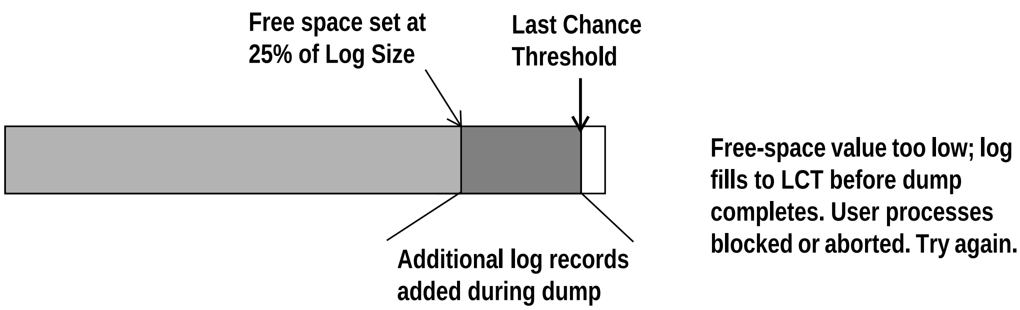


Figure 10-6: Additional Log Threshold Doesn't Begin Dump Early Enough

25% free space is not enough. Try initiating the dump transaction when the log has 37.5% fee space:

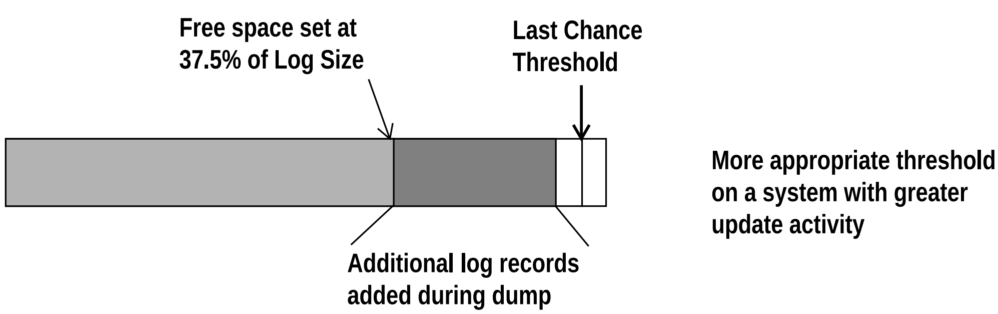


Figure 10-7: Moving Threshold Leaves Enough Free Space to Complete Dump

Use **sp_modifythreshold** to change the free-space value to 37.5% of log capacity. For example:

```
sp_modifythreshold mydb, logsegment, 768,
     thresh_proc
```

Creating Additional Thresholds on Other Segments

You can also create thresholds on data segments. For example, you might create a threshold on the default segment used to store tables and indexes. You would also create an associated stored procedure to print messages in your error log when space on the *default* segment falls below this threshold. If you monitor the error log for these messages, you can add space to the database device when you need it—before your users encounter problems.

The following example creates a threshold on the *default* segment of *mydb*. When the free space on this segment falls below 200 pages, SQL Server executes the procedure *space_dataseg*:

```
sp_addthreshold mydb, "default", 200, space_dataseg
```

Determining Threshold Placement

Each new threshold must be at least 2 times *@@thresh_hysteresis* pages from the next closest threshold:

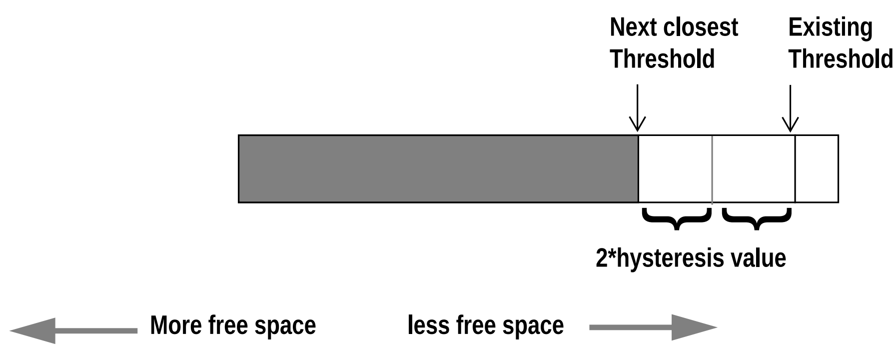


Figure 10-8: Determining Where to Place a Threshold

Use this command:

```
select @@thresh_hysteresis
```

to see the hysteresis value for a database.

In the following example, a segment has a threshold set at 100 pages. The hysteresis value for the database is 64 pages. The next threshold must be at least 100 + (2 * 64), or 228 pages.

```
select @@thresh_hysteresis
-----------
         64
sp_addthreshold user_log_dev, 228,
    sp_thresholdaction
```

Creating Threshold Procedures

Sybase does not supply **sp_thresholdaction** or other threshold procedures. You must create these procedures yourself to ensure that they are tailored to your site's needs.

Suggested actions for **sp_thresholdaction** include writing to the server's error log and dumping the transaction log to increase the amount of log space. You can also design the threshold procedure to execute remote procedure calls to an Open Server, sending mail to the appropriate individuals when a threshold is crossed.

This section provides some guidelines for writing threshold procedures. It includes two sample procedures.

Declaring Procedure Parameters

SQL Server passes four parameters to a threshold procedure:

- *@dbname, varchar(30),* which contains the database name.
- *@segmentname, varchar(30),* which contains the segment name.
- *@space_left, int,* which contains the space-left value for the threshold.
- *@status, int,* which has a value of 1 for last-chance thresholds and 0 for other thresholds.

These parameters are passed by position rather than by name. Your procedure can use other names for these parameters, but must declare them in the order shown and with the datatypes shown.

Generating Error Log Messages

Executing a threshold procedure does not automatically generate any log messages: if your procedure does not contain a **print** or **raiserror** statement, the error log will not contain any record of the threshold event. You should include a **print** statement near the beginning of your procedure to record the database name, segment name, and threshold size in the error log.

The process that executes threshold procedures is an internal SQL Server process. It does not have an associated user terminal or network connection. If you test your threshold procedures by executing them directly (that is, using **execute *procedure_name***) during a terminal session, you see the output from the **print** and **raiserror** messages on your screen. When the same procedures are executed by reaching a threshold, the messages go to the error log. The messages in the log include the date and time.

For example, if **sp_thresholdaction** includes this statement:

```
print "LOG DUMP: log for '%1!' dumped", @dbname
```

SQL Server writes this message to the error log:

```
00: 92/09/04 15:44:23.04 server: background task message: LOG
DUMP: log for 'pubs2' dumped
```

Dumping the Transaction Log

If your **sp_thresholdaction** procedure includes a **dump transaction** command, SQL Server dumps the log to the devices named in the procedure. The **dump transaction** command truncates the transaction log by removing all pages from the beginning of the log, up to the page just before the page that contains an uncommitted transaction record

Once there is enough log space, suspended transactions are awakened. If you abort transactions rather than suspending them, users must resubmit them.

Generally, dumping to a disk is **not** recommended, especially to a disk that is on the same machine or the same disk controller as the database disk. However, since threshold-initiated dumps can take place at any time, you may wish to dump to disk and then copy the resulting files to off-line media. (You will have to copy the files back to the disk in order to reload them.)

Your choice will depend on:

- Whether you have a dedicated dump device online, loaded and ready to receive dumped data
- Whether you have operators available to mount tape volumes during all times when your database is available
- The size of your transaction log
- Your transaction rate
- Your regular schedule for dumping databases and transaction logs
- Available disk space
- Other site-specific dump resources and constraints

A Simple Threshold Procedure

Following is a simple procedure that dumps the transaction log and prints a message to the error log. Because this procedure uses a variable (*@dbname*) for the database name, it can be used for all databases on a SQL Server:

```
create procedure sp_thresholdaction
    @dbname varchar(30),
    @segmentname varchar(30),
    @free_space int,
    @status int
as
dump transaction @dbname
    to tapedump1
print "LOG DUMP: '%1! for '%2!' dumped",
        @segmentname, @dbname
```

A More Complex Procedure

The next threshold procedure performs different actions, depending on the value of the parameters passed to it. Its conditional logic allows it to be used with both log and data segments.

This procedure:

- Prints a "LOG FULL" message if the procedure was called as the result of reaching the log's last-chance threshold. The status bit is 1 for the last-chance thresholds and 0 for all other thresholds. The test **if (@status&1) = 1** returns a value of true only for the last-chance threshold.
- Verifies that the segment name provided is the log segment. The segment ID for the log segment is always 2, even if the name has been changed.
- Prints "before" and "after" size information on the transaction log. If the log did not shrink significantly, a long-running transaction may be causing the log to fill.
- Prints the time the transaction log dump started and stopped, helping gather data about dump durations.
- Prints a message in the error log if the threshold is not on the log segment. The message gives the database name, the segment name and the threshold size, letting you know that the data segment of a database is filling up.

```
create procedure sp_thresholdaction
    @dbname         varchar(30),
    @segmentname    varchar(30),
    @space_left     int,
    @status         int
as
declare @devname varchar(100),
        @before_size int,
        @after_size int,
        @before_time datetime,
        @after_time datetime,
        @error int

/*
** if this is a last-chance threshold, print a LOG FULL msg
** @status is 1 for last-chance thresholds,0 for all others
*/
if (@status&1) = 1
begin
    print "LOG FULL: database ’%1!’", @dbname
end

/*
** if the segment is the logsegment, dump the log
** log segment is always "2" in syssegments
*/
if @segmentname = (select name from syssegments
            where segment = 2)
begin
    /* get the time and log size
    ** just before the dump starts
    */
    select  @before_time = getdate(),
        @before_size = reserved_pgs(id, doampg)
    from sysindexes
    where sysindexes.name = "syslogs"

    print "LOG DUMP: database ’%1!’, threshold ’%2!’",
        @dbname, @space_left

select @devname = "/backup/" + @dbname + "_" +
        convert(char(8), getdate(),4) + "_" +
        convert(char(8), getdate(), 8)

dump transaction @dbname to @devname

    /* error checking */
    select @error = @@error
    if @error != 0
```

```
    begin
        print "LOG DUMP ERROR: %1!", @error
    end

/* get size of log and time after dump */
    select @after_time = getdate(),
        @after_size = reserved_pgs(id, doampg)
        from sysindexes
        where sysindexes.name = "syslogs"

    /* print messages to error log */
    print "LOG DUMPED TO: device ’%1!", @devname
    print "LOG DUMP PAGES: Before: ’%1!’, After ’%2!’",
        @before_size, @after_size
    print "LOG DUMP TIME: %1!, %2!", @before_time, @after_time
end     /* end of ’if segment = 2’ section */
else    /* this is a data segment, print a message */
begin
    print "THRESHOLD WARNING: database ’%1!’, segment ’%2!’ at
’%3!’ pages", @dbname, @segmentname, @space_left
end
```

Deciding Where to Put a Threshold Procedure

Although you can create a separate procedure to dump the transaction log for each threshold, it is far easier to create a single threshold procedure that is executed by all of your log segment thresholds. When the amount of free space on a segment falls below a threshold, SQL Server reads the *systhresholds* table in the affected database for the name of the associated stored procedure. The entry in *systhresholds* can specify any of the following:

- A remote procedure call to an Open Server
- A procedure name qualified by a database name (for example, *sybsystemprocs.dbo.sp_thresholdaction*)
- An unqualified procedure name

If the procedure name does not include a database qualifier, SQL Server looks in the database where the shortage of space occurred. If it cannot find the procedure there, and if the procedure name begins with the characters “sp_”, SQL Server looks for the procedure in the *sybsystemprocs* database and then in *master* database.

If SQL Server cannot find the threshold procedure, or cannot execute it, it prints a message in the error log.

Disabling Free Space Accounting for Data Segments

Use the **no free space acctg** option to **sp_dboption**, followed by the **checkpoint** command, to disable free space accounting on non-log segments. You **cannot** disable free-space accounting on the log segment.

When you disable free space accounting, only the thresholds on your log segment monitor space usage. Crossing thresholds on your data segments will not cause your threshold procedures to be executed. Disabling free space accounting speeds recovery time because free-space counts are not recomputed during recovery for any segment except the log segment.

The following example turns off free-space accounting for the *production* database:

```
sp_dboption production,
    "no free space acctg", true
```

◆ ***WARNING!***

System procedures cannot provide accurate information about space allocation when free-space accounting is disabled.

Creating a Last-Chance Threshold for Existing Databases

When you upgrade an existing database, it does not automatically acquire a last-chance threshold. Use the **lct_admin** system function in existing databases to create a last-chance threshold. Its syntax is:

```
select lct_admin ("lastchance", db_id())
```

➤ ***Note***

Only databases that store their logs on a separate segment can have a last-chance threshold. See "Moving the Transaction Log to Another Device" on page 3-28 for more information.

Special Topics

11 Diagnosing System Problems

Introduction

This chapter discusses diagnosing and fixing system problems. Topics covered include:

- How SQL Server responds to system errors
- The SQL Server and Backup Server error logs
- Error messages
- Types of error messages (severity levels)
- Killing SQL Server processes

How SQL Server Responds to System Problems

When SQL Server encounters a problem—whether caused by the user or the system—it displays information about the problem, how serious it is, and what you can do to fix it. This information consists of:

- A message number, which uniquely identifies the error message
- A severity level number between 10 and 24
- An error state number, which allows unique identification of the line of SQL Server code at which the error was raised
- An error message, which tells you what the problem is, and which may suggest how to fix it

For example, here's what happens if you make a typing error and try to access a table that doesn't exist:

```
select * from pulbishers
```

```
Msg 208, Level 16, State 1:
Invalid object(table) name pulbishers
```

In some cases, there can be more than one error message for a single query. If there is more than one error in a batch or query, SQL Server usually reports only the first one. Subsequent errors are caught the next time you execute the batch or query.

The error messages are stored in *master..sysmessages*, which is updated with each new release of SQL Server. Here are the first few rows (from a server with *us_english* as the default language):

```
select error, severity, description
from sysmessages
where error >=101 and error <=106
and langid is null
```

```
101 15 Line %d: SQL syntax error.
102 15 Incorrect syntax near '%.*s'.
103 15 The %S_MSG that starts with '%.*s' is too long. Maximum
       length is %d.
104 15 Order-by items must appear in the select-list if the
       statement contains set operators.
105 15 Unclosed quote before the character string '%.*s'.
106 16 Too many table names in the query. The maximum allowable
       is %d.

(6 rows affected)
```

You can generate your own list by querying *sysmessages*. Here is some additional information for writing your query:

- If your server supports more than one language, *sysmessages* stores each message in each language. The column *langid* is NULL for us_english and matches the *syslanguages.langid* for other languages installed on the server. For information about languages on your server, use **sp_helplanguage.**
- The *dlevel* column in *sysmessages* is currently unused.
- The *sqlstate* column stores the SQLSTATE value for error conditions and exceptions defined in ANSI SQL '92.
- Message numbers 17000 and greater are system procedure error messages and message strings.

Error Messages and Message Numbers

The combination message number (*error*) and the language ID (*langid*) uniquely identifies each error message. Messages with the same message number, but a different language IDs are translations.

```
select error, description, langid
from sysmessages
where error = 101
```

```
error description                            langid
----- -------------------------------------- ------
  101 Line %d: SQL syntax error.               NULL
  101 Ligne %1!: erreur de syntaxe SQL.           1
  101 Zeile %1!: SQL Syntaxfehler.                2

(3 rows affected)
```

The error message text is a description of the problem. The descriptions often include a line number, a reference to a kind of database object (a table, column, stored procedure, etc.), or the name of a particular database object.

In the *description* field of *sysmessages*, a percent symbol (%) followed by a character or character string serves as a place holder for these pieces of data, which SQL Server supplies when it encounters the problem and generates the error message. “%d” is a place holder for a number; “%S_MSG” is a place holder for a kind of database object; “%.*s”—all within quotes—is a place holder for the name of a particular database object. Figure 11-1 lists place holders and what they represent.

For example, the *description* field for message number 103 is:

```
The %S_MSG that starts with ’%.*s’ is too long.
Maximum length is %d.
```

The actual error message as displayed to a user might be:

```
The column that starts with ’title’ is too long.
Maximum length is 80.
```

For errors that you report, it is important to include the numbers, object types, and object names. (See the section ‘‘Reporting Errors’’ on page 11-11.)

Variables in Error Message Text

The following chart explains the abbreviations that appear in the error message text provided with each error message explanation:

Symbol	Stands For
%d,%D	a decimal number
%x, %X, %.*x, %lx, %04x, %08lx	a hexadecimal number

Table 11-1: Error Text Abbreviation Key

Symbol	Stands For
%s	a null-terminated string
%.*s, %*s, %*.s	a string, usually the name of a particular database object
%S_*type*	a SQL Server-defined structure
%c	a single character
%f	a floating point number
%ld	a long decimal
%lf	a double floating point number

Table 11-1: Error Text Abbreviation Key

Error Logging

Most error messages from SQL Server go only to the user's screen.

The backtrace from fatal error messages (severity levels 19 and higher), and error messages from the kernel are sent to an error log file. The name of this file varies; see the *System Administration Guide Supplement* for your platform or the *SQL Server Utility Programs* manual.

➤ ***Note***

The error log file is owned by the user who installed SQL Server (or to the person who started the server after an error log was removed). Permissions or ownership problems with the error log at the operating system level can block successful startup of SQL Server.

SQL Server creates an error log for you if it does not already exist. The location of this log depends on how you start the server:

- If you use **startserver** and the "runserver" file created by the SQL Server installation process, the **dataserver** command includes a flag that specifies the location of the error log.
- If you type the **dataserver** command (or **sqlsrvr** for PC platforms) from the operating system, you can include the flag that specifies the location of the errorlog (**-e** for UNIX and PC platforms; **/errorfile** in OpenVMS).
- If you do not include the error log flag, the error log is created in the directory where you start SQL Server.

➤ ***Note***

You should always restart SQL Server from the same directory, or with the "runserver" file, or the error log flag, so that you can locate your error logs.

Each time you start a particular server, boot messages in the error log provide information on the success (or failure) of the start and the recovery of each database on the server. Subsequent fatal error messages and all kernel error messages are appended to the error log file. If you need to reduce the size of the error log (throw out old or unneeded messages) you must prune the log while SQL Server is shut down.

Error Log Format

Entries in the error log include the following information:

- The engine involved for each log entry. The engine number is indicated by a two digit number followed by a colon at the beginning of each line.
- The date, displayed in this format: *yy/mm/dd,* which allows you to sort error messages by date.
- The time, displayed in 24-hour format. The time format includes seconds and hundredths of a second.
- The error message itself.

The following is an example of a line from the error log:

```
00: 94/02/18 13:40:28.13 server: Recovery complete.
```

The entry "00:" at the beginning of the line shows that the message was produced by engine number 0. The date is February 18, 1994, and the time is 1:40 p.m., 28 seconds, and 13 hundredths of a second.

Severity Levels

The severity level provides information about the kind of problem SQL Server has encountered. For maximum integrity, when SQL Server responds to error conditions it displays messages from *sysmessages* but takes action according to an internal table. Because a few corresponding messages differ in severity levels, occasionally you may notice a difference in expected behavior if you are developing applications or procedures that refer to SQL Server messages and severity levels.

◆ ***WARNING!***

You can create your own error numbers and messages based on SQL Server error numbers (say by adding 20,000 to the SQL Server value), but do not alter the SQL Server-supplied system messages in *sysmessages*.

Add user-defined error messages to *sysusermessages* with the stored procedure **sp_addmessage**. See Volume 2 of the *SQL Server Reference Manual* for more information.

In **isql** and Data Workbench, severity level 10 ("status" or "informational") messages do not display their message number or severity level. In Open Client applications, SQL Server returns "0" for severity level 10.

Severity levels through 16 indicate problems caused by mistakes in what users have entered.

Severity levels over 16 indicate software or hardware errors. If the number is 17 or 18, you'll be able to continue the work you're doing (though you may not be able to execute a particular command).

Problems with severity levels 19 or over are system problems. They are fatal errors, which means that the process (the program code that is running in order to accomplish the task that you specified in your command) is no longer running. The process freezes its state before it stops, recording information about what was happening. It is then killed and disappears.

These errors break the user's connection to SQL Server. Depending on the problem, a user may or may not be able to reconnect and resume working. Some problems with severity levels in this range affect only one user and one process. Others affect all the processes in the database. In some cases, it will be necessary to restart SQL Server. These problems don't necessarily damage a database or its objects, but they can. They may also result from earlier damage to a database or its objects. Still other problems are caused by hardware malfunctions.

A backtrace of fatal error messages from the kernel is directed to the error log file, where the System Administrator can review it.

Users should be instructed to inform the System Administrator whenever problems that generate severity levels of 17 and over occur. The System Administrator is responsible for resolving them and tracking their frequency. The System Administrator should monitor all problems that generate severity levels of 17 through 24.

If the problem has affected an entire database, the System Administrator may have to use the Database Consistency Checker (**dbcc**) to determine the extent of the damage. The **dbcc** may identify some objects that have to be removed. It can repair some damage, but the database may also have to be reloaded. **dbcc** is discussed in more detail in Chapter 6, "Checking Database Consistency". Loading a user database is discussed in Chapter 8, "Backing Up and Restoring User Databases"; loading system databases is discussed in Chapter 9, "Backing Up and Restoring the System Databases".

The next subsections discuss each severity level.

Levels 10 Through 18

Error messages with severity levels 10 through 16 are generated by problems caused by user errors, and can always be corrected by the user. Severity levels 17 and 18 do not terminate the user's session.

Error messages with severity levels 17 or higher should be reported to the System Administrator or Database Owner.

Level 10: Status Information

Messages with severity level 10 are not errors at all. They provide additional information after certain commands have been executed and typically do not display the message number or severity level. For example, after a **create database** command has run, SQL Server displays a message telling the user how much space has actually been allocated for the new database.

Level 11: Specified Database Object Not Found

Messages with severity level 11 indicate that SQL Server can't find an object referenced in the command.

This is often because the user has made a mistake in typing the name of a database object, because the user did not specify the object owner's name, or because of confusion about which database is current. Check spelling of object names, use owner names if the object is not owned by you or "dbo", and make sure you're in the correct database.

Level 12: Wrong Datatype Encountered

Messages with severity level 12 indicate a problem with datatypes. For example, the user may have tried to enter a value of the wrong datatype into a column, or to compare columns of different (and incompatible) datatypes.

To correct comparison problems, use the **convert** function with **select**. For information on **convert**, see Volume 1 of the *SQL Server Reference Manual* or the *Transact-SQL User's Guide*.

Level 13: User Transaction Syntax Error

Messages with severity level 13 indicate that something is wrong with the current user-defined transaction. For example, you may have issued a **commit transaction** command without having issued a **begin transaction**, or you may have tried to roll a transaction back to a savepoint that has not been defined (sometimes there may be a typing or spelling mistake in the name of the savepoint).

Severity level 13 can also indicate a deadlock, in which case the deadlock victim's process is rolled back. The user must restart his or her command.

Level 14: Insufficient Permission to Execute Command

Messages with severity level 14 mean you don't have the permission necessary to execute the command or access the database object. You can ask the owner of the database object, the owner of the database, or the System Administrator to grant you permission to use the command or object in question.

Level 15: Syntax Error in SQL Statements

Messages with severity level 15 indicate that the user has made a mistake in the syntax of the command. The text of these error messages includes the line numbers on which the mistake occurs, and the specific word near which it occurs.

Level 16: Miscellaneous User Error

Error messages with severity level 16 reflect that the user has made some kind of non-fatal mistake that doesn't fall into any of the other categories.

For example, the user may have tried to update a view in a way that violates the restrictions. Another error that falls into this category is unqualified column names in a command that includes more than one table with that column name. SQL Server has no way to determine which one the user intends. Check command syntax and working database context.

➤ ***Note***

The System Administrator should monitor occurrences of errors with severity levels 17 through 24. Levels 17 and 18 are usually not reported in the error log. Users should be instructed to notify the System Administrator when errors of level 17 and 18 occur.

Level 17: Insufficient Resources

Error messages with severity level 17 mean that the command has caused SQL Server to run out of resources (usually space for the database on the disk) or to exceed some limit set by the System Administrator.

These system limits include the number of databases that can be open at the same time and the number of connections allowed to SQL Server. They are stored in system tables, and can be checked with the **sp_configure** command. See Chapter 12, ‘‘Fine-Tuning Performance and Operations’’, for more information on changing configuration variables.

The Database Owner can correct level 17 error messages indicating that you have run out of space. Other level 17 error messages should be corrected by the System Administrator.

Level 18: Non-Fatal Internal Error Detected

Error messages with severity level 18 indicate some kind of internal software bug. However, the command runs to completion, and the connection to SQL Server is maintained. An example of a situation that generates severity level 18 is SQL Server detecting that a decision about the access path for a particular query has been made without a valid reason.

Since problems that generate such messages don’t keep users from their work, they may have a tendency not to report them. Users should be instructed to inform the System Administrator every time an error message with this severity level (and higher levels) occurs, so that the System Administrator can report them.

Severity Levels 19 Through 24

Fatal problems generate error messages with severity levels 19 and higher. They break the user’s connection to SQL Server. To continue working, the user must restart the client program.

Level 19: SQL Server Fatal Error in Resource

Error messages with severity level 19 indicate that some non-configurable internal limit has been exceeded, and that SQL Server cannot recover gracefully. You must reconnect to SQL Server. See your System Administrator.

Level 20: SQL Server Fatal Error in Current Process

Error messages with severity level 20 mean that SQL Server has encountered a bug in some command. The problem has affected only the current process; it is unlikely that the database itself has been damaged. Run **dbcc** diagnostics. You must reconnect to SQL Server. See your System Administrator.

Level 21: SQL Server Fatal Error in Database Processes

Error messages with severity level 21 mean that SQL Server has encountered a bug that affects all the processes in the current database. However, it is unlikely that the database itself has been damaged. Restart SQL Server and run the **dbcc** diagnostics. You must reconnect to SQL Server. See your System Administrator.

Level 22: SQL Server Fatal Error: Table Integrity Suspect

Error messages with severity level 22 mean that the table or index specified in the message has been damaged at some previous time by a software or hardware problem.

The first step is to restart SQL Server and run **dbcc** to determine if other objects in the database are also damaged. Whatever the report from **dbcc**, it's possible that the problem is in the cache only, and not on the disk itself. If so, restarting SQL Server will fix the problem.

If restarting doesn't help, the problem is on the disk as well. Sometimes the problem can be solved by dropping the object specified in the error message. For example, if the message tells you that SQL Server has found a row with length 0 in a nonclustered index, the table owner can drop the index and re-create it.

You must reconnect to SQL Server. See your System Administrator.

Level 23: SQL Server Fatal Error: Database Integrity Suspect

Error messages with severity level 23 mean that the integrity of the entire database is suspect due to damage caused at some previous time by a software or hardware problem. Restart SQL Server and run the **dbcc** diagnostics.

Even when a level 23 error indicates that the whole database is suspect, the damage may be confined to the cache, and the disk itself may be fine. If so, restarting SQL Server with **startserver** will fix the problem.

Level 24: Hardware Error or System Table Corruption

Error messages with severity level 24 reflect some kind of media failure or (in rare cases) the corruption of *sysusages*. The System Administrator may have to reload the database. It may be necessary to call your hardware vendor.

Reporting Errors

When you report an error be sure to include the following information:

- The message number, level number, and state number.
- Any numbers, database object types, or database object names that are included in the error message.
- The context in which the message was generated—which command was running at the time. You can help by providing a hard copy of the backtrace from the error log.

Backup Server Error Logging

Like SQL Server, Backup Server creates an error log for you if it does not already exist. The location of this log depends on how you start the server:

- If you use **startserver** and the "runserver" file created by the installation process, the **backupserver** command includes a flag that specifies the location of the error log
- If you type the **backupserver** command from the operating system, you can include the flag that specifies the location of the errorlog (**-e** for UNIX and PC platforms; **/errorfile** in OpenVMS)
- If you do not include the error log flag, the error log is created in the directory where you start Backup Server.

Backup Server error message are in the form:

```
MMM DD YYY: Backup Server:N.N.N.N: Message Text
```

Backup Server message numbers consist of 4 integers separated by periods, in the form N.N.N.N. Messages in the form N.N.N are sent by Open Server.

The four components of a Backup Server error message are *major.minor.severity.state*.

- The *major* component generally indicates the functional area of the Backup Server code where the error occurred:
  - 1 - System errors
  - 2 - Open Server event errors.
  - 3 - Backup Server remote procedure call errors.
  - 4 - I/O service layer errors.
  - 5 - Network data transfer errors.
  - 6 - Volume handling errors.
  - 7 - Option parsing errors.

  Major error categories 1-6 may result from Backup Server internal errors or a variety of system problems. Major errors in category 7 are almost always due to problems in the options you specified to your dump or load command.
- *minor* numbers are assigned in order within a major category.
- *severity* is one of the following:
  - 1 - Informational, no user action necessary.
  - 2, 3 - An unexpected condition, possibly fatal to the session, has occurred. Error may have occurred with any or all of usage, environment or internal logic.
  - 4 - An unexpected condition, fatal to the execution of the Backup Server, has occurred. The Backup Server must exit immediately.
- *state* codes have a one-to-one mapping to instances of the error report within the code. If you need to contact Technical Support about Backup Server errors, the state code helps determine the exact cause of the error.

Killing Processes

A process is a task being carried out by SQL Server. Processes can be initiated by a user giving a command, or by SQL Server itself. Each process is assigned a unique process identification number when it starts. These ID numbers, and other information about each process, are stored in *master..sysprocesses*. You can see most of the information by running the system procedure **sp_who**.

The **kill** command gets rid of an ongoing process. The most frequent reason for killing a process is that it interferes with other users and the person responsible for running it isn't available. The process may hold locks that block access to database objects, or there may be many sleeping processes occupying the available user connections. A System Administrator can kill processes that are:

- Waiting for an alarm, such as a **waitfor** command
- Waiting for network sends or receives
- Waiting for a lock
- Most running or runnable processes

SQL Server allows you to kill processes only if it can cleanly roll back any uncompleted transactions and release all system resources that are used by the process.

Running **sp_who** on a single-engine server shows the **sp_who** process "running" and all other processes "runnable" or in one of the sleep states. In multi-engine servers, there can be a "running" process for each engine.

The following table shows the values that **sp_who** reports:

Status	Condition	Effects 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 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 Server reboot to clear
`runnable`	in the queue of runnable processes	immediate

Table 11-2: Status Values Reported by sp_who

Status	Condition	Effects of kill Command
`running`	actively running on one on the Server engines	immediate
`infected`	Server has detected serious error condition; extremely rare	**kill** command not recommended. Server reboot probably required to clear process
`background`	a process, such as a threshold procedure, run by SQL 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	killed when it "wakes up": 1) when space is freed in the log by a **dump transaction** command or 2) when an SA uses the **lct_admin** function to wake up "log suspend" processes

Table 11-2: Status Values Reported by sp_who (continued)

Only a System Administrator can issue the **kill** command: permission to use it cannot be transferred.

The syntax is:

```
kill spid
```

You can kill only one process at a time. A **kill** command is not reversible, and cannot be included in a user-defined transaction. *spid* must be a numeric constant; you cannot use a variable. Here is some sample output from **sp_who**:

```
spid status    loginame hostname blk dbname cmd
 ---- --------  -------- -------- --- ------ -------------
   1 recv sleep  bird     jazzy    0   master AWAITING COMMAND
   2 sleeping    NULL              0   master NETWORK HANDLER
   3 sleeping    NULL              0   master MIRROR HANDLER
   4 sleeping    NULL              0   master AUDIT PROCESS
   5 sleeping    NULL              0   master CHECKPOINT SLEEP
   6 recv sleep  rose     petal    0   master AWAITING COMMAND
   7 running     sa       helos    0   master SELECT
   8 send sleep  daisy    chain    0   pubs2  SELECT
   9 alarm sleep lily     pond     0   master WAITFOR
  10 lock sleep  viola    cello    7   pubs2  SELECT
```

Processes 2-5 cannot be killed: they are system processes. The login name NULL and the lack of hostname identify them as system processes. You will always see NETWORK HANDLER, MIRROR HANDLER and CHECKPOINT SLEEP (or rarely, CHECKPOINT). AUDIT PROCESS becomes activated if you enable auditing.

Processes 1, 6, 8, 9 and 10 can be killed, since they have the status values "recv sleep", "send sleep", "alarm sleep" and "lock sleep".

In **sp_who** output, you cannot tell whether a process whose status is "recv sleep" belongs to a user who is using SQL Server, but may be pausing to examine the results of a command, or whether the process indicates that a user has rebooted a PC or other terminal, and left a stranded process.You can learn more about a questionable process by querying the *sysprocesses* table for information. For example, this query shows the host process ID and client software used by process 8:

```
select hostprocess, program_name
    from sysprocesses
where spid = 8
```

```
 hostprocess program_name
 ----------- ----------------
 3993        isql
```

This query, plus the information about the user and host from the **sp_who** results, provides additional information for tracking down the process from the operating system level.

Using *sp_lock* to Examine Blocking Processes

In addition to **sp_who**, described above, the system procedure **sp_lock** can help identify processes that are blocking other processes. If the *blk* column in the **sp_who** report indicates that another process is blocked waiting to acquire locks, **sp_lock** can display information about the blocking process. For example, process 10 in the **sp_who** output above is blocked by process 7. To see information about process 7, execute:

```
sp_lock 7
```

For more information about locking in SQL Server, see Chapter 13, "Locking".

Shutting Down Servers

A System Administrator can shut down SQL Server or a Backup Server with the **shutdown** command. The syntax is:

```
shutdown [backup_server_name] [with {wait|nowait}]
```

The default for the **shutdown** command is **with wait**, that is, **shutdown** and **shutdown with wait** do exactly the same thing.

Shutting Down SQL Server

If you do not give a server name, **shutdown** shuts down the SQL Server you are using. When you issue a **shutdown** command, SQL Server:

- Disables logins, except for System Administrators
- Performs a checkpoint in each database, flushing pages that have changed from memory to disk
- Waits for currently executing SQL statements or procedures to finish

This minimizes the amount of work that automatic recovery must do when you restart SQL Server.

The **with no_wait** option shuts down SQL Server immediately. User processes are aborted, and recovery may take longer after a **shutdown with nowait**. You can help minimize recovery time by issuing a **checkpoint** command before you issue a **shutdown with nowait** command.

Shutting Down a Backup Server

To shut down a Backup Server, give the Backup Server's name:

```
shutdown SYB_BACKUP
```

The default is **with wait**, so any dumps or loads in progress will complete before the Backup Server process halts. Once you issue a **shutdown** command, no new dump or load sessions can be started on the Backup Server.

To see the names of Backup Servers accessible from your SQL Server, execute **sp_helpserver.** Use the value in the *name* column in the **shutdown** command. You can only shut down a Backup Server that is:

- Listed in *sysservers* on your SQL Server, and
- Listed in your local interfaces file.

Use **sp_addserver** to add a Backup Server to *sysservers*.

Checking for Active Dumps and Loads

To see the activity on your Backup Server before executing a shutdown command, run **sp_who** on the Backup Server:

```
SYB_BACKUP...sp_who
```

```
spid   status   loginame hostname   blk cmd
------ -------- -------- ---------- --- ----------------
     1 sleeping NULL     NULL       0   CONNECT HANDLER
     2 sleeping NULL     NULL       0   DEFERRED HANDLER
     3 runnable NULL     NULL       0   SCHEDULER
     4 runnable NULL     NULL       0   SITE HANDLER
     5 running  sa       heliotrope 0   NULL
```

Using nowait on a Backup Server

The **shutdown *backup_server* with nowait** command shuts down the Backup Server, regardless of current activity. Use it only in severe circumstances. It can leave your dumps or loads in incomplete or inconsistent states.

If you use **shutdown...with nowait** during a log or database dump, check for the message indicating that the dump completed. If you did not receive this message, or you are uncertain, your next dump should be a **dump database**, not a transaction dump. This guarantees that you will not be relying on possibly inconsistent dumps.

If you use **shutdown with nowait** during a load of any kind, and you did not receive the message indicating that the load completed, you may not be able to issue further **load transaction** commands on the database. Be sure to run a full database consistency check (**dbcc**) on the database before you use it. You may have to re-issue the full set of load commands, starting with **load database.**

12 Fine-Tuning Performance and Operations

Introduction

The System Administrator or Database Owners may want to allocate SQL Server resources to optimize performance. In part, this may involve being able to estimate the size of tables and indexes in a database, and the placement of these objects on separate physical disks.

Once your application is up and running, the System Administrator monitors its performance, and may want to customize and fine-tune it. SQL Server provides software tools for these purposes. This chapter explains:

- Setting query processing options with the **set** command
- Setting database options with the system procedure **sp_dboption**
- Monitoring SQL Server activity with **sp_monitor**
- Using **update statistics** to ensure that SQL Server makes the best use of existing indexes
- Changing system variables with **sp_configure** and the **reconfigure** command
- Placing objects on segments to improve performance
- Estimating the size of database objects

Tuning Queries and Stored Procedures

The query processing options allow you to instruct SQL Server to handle queries and stored procedures in a variety of unusual ways. They are turned on or off for the duration of the user's work session with the Transact-SQL **set** command. No permissions are required for using the **set** command.

The **set** command syntax allows certain options to be grouped in one command. The groups are:

```
set ansinull  {on | off}

set ansi_permissions {on | off}

set arithabort [overflow | numeric_truncation]
   {on | off}

set arithignore [overflow] {on | off}
```

```
set {chained, close on endtran, nocount, noexec,
   parseonly, procid, self_recursion, showplan}
   {on | off}

set char_convert {off | on [with {error | no_error}] |
   charset [with {error | no_error}]}

set cursor rows number for cursor_name

set {datefirst number, dateformat format,
   language language}

set dup_in_subquery {on | off}

set fipsflagger {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 quoted_identifier {on | off}

set role {"sa_role" | "sso_role" | "oper_role"}
   {on | off}

set {rowcount number, textsize number}

set statistics {io, time} {on | off}

set string_rtruncation {on | off}

set transaction isolation level {1 | 3}
```

For example, this statement causes SQL Server to return a description of the processing plan for each query, but not execute it:

```
set showplan, noexec on
```

This statement causes SQL Server to stop processing each query after it returns the first ten rows:

```
set rowcount 10
```

See Volume 1 of the *SQL Server Reference Manual* for more information and examples.

Here are brief explanations of the **set** options (in alphabetic order):

- **ansinull** – determines whether or not evaluation of NULL-valued operands in SQL equality (=) or inequality (!=) comparisons or aggregate functions, also called set functions, is ANSI-compliant. By default **ansinull** is set **off**. This option does not affect how SQL Server evaluates NULL values in other kinds of SQL statements, such as **create table**.

- **ansi_permissions** – determines what types of permissions SQL Server requires to execute **update** and **delete** statements. By default **ansi_permissions** is set **off**. **update** statements require **update** permission on the columns being changed. **delete** statements require **delete** permission on the table.

  Turn **ansi_permissions on** to perform additional permissions checks required for ANSI compliance. In addition to the existing permissions requirements, both **update** and **delete** statements require **select** permission on all columns in the **where** clause; **update** statements require **select** permission on columns on the right side of the **set** clause.

- **arithabort** – determines how SQL 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 behavior following a divide-by-zero error or a loss of precision during either an explicit or an implicit datatype conversion. This type of error is considered serious. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that causes the error but continues to process other statements in the transaction or batch.
  - **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**, SQL Server truncates the query results and continues processing.

- **arithignore arith_overflow** – determines whether SQL Server displays a message after a divide-by-zero error or a loss of precision. The default setting, **off**, displays a warning message after these errors. Setting **arithignore arith_overflow on** suppresses warning messages after these errors. The optional **arith_overflow** keyword can be omitted without any effect.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for Release 10.0. If your application is based on a previous SQL Server release use these options, examine them to be sure they still produce the desired effect.

- **chained** – begins a transaction just before the first data retrieval or modification statement at the beginning of a session and after a transaction ends. In chained mode, SQL Server implicitly executes a **begin transaction** before the following statements: **delete**, **fetch**, **insert**, **open**, **select**, and **update**. You cannot execute **set chained** within a transaction.
- **char_convert** – turns off and on character set conversion between SQL Server and a client. See Chapter 13 for a full discussion of this option.
- **close on endtran** – causes SQL Server to close all cursors opened within a transaction at the end of that transaction. A transaction ends by using either the Transact-SQL **commit** or the **rollback** statement.
- **cursor rows** – causes SQL Server to return the *number* of rows for each cursor **fetch** request from a client application. You can **set** the **cursor rows** option for a cursor whether it is open or closed.
- **datefirst** – sets the first weekday to a number from one through seven. The us_english 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*, or *dym*. The us_english default is *mdy*.
- **dup_in_subquery** – controls whether a subquery using an **in** clause returns duplicate values. The default is **off**; duplicate rows are not returned. Prior to Release 10.0, SQL Server returned a row for each matching row in the subquery. ANSI specifies removing duplicate values from the result set. This option is provided as a workaround until you can re-write your queries for 10.0 behavior; it will not be supported in future releases.
- **fipsflagger** – determines whether SQL Server displays a warning message when Transact-SQL extensions to SQL are used. By default, SQL Server does not tell you when you use non-ANSI SQL.
- **identity_insert** – determines whether 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 on** allows the table owner, Database Owner, or a System Administrator to insert a value into the IDENTITY column.

- **language** – is the official name of the language that displays system messages. The language must be available on the server. The **sybinit** installation program sets the default language.
- **nocount** – turns off the display of the number of "rows affected" from a query that returns rows. The global variable *@@rowcount* is updated even when **nocount** is on.
- **noexec** – compiles each query into a query tree but does not execute it. **noexec** is often used with **showplan** and **statistics io**. Once **noexec** is turned on, no subsequent commands are executed (including other **set** commands) until **noexec** is turned off.
- **offsets** – returns the offset (position in relation to the beginning of the query) of specified keywords in Transact-SQL statements. The keyword list is a comma-separated list that can include any of the following Transact-SQL reserved words: **select**, **from**, **order**, **compute**, **table**, **procedure**, **statement**, **param**, and **execute**. This option is meant to be used in Open Client DB-Library.
- **parseonly** – checks the syntax of each query and returns any error messages, without generating a sequence tree, compiling, or executing the query. It returns offsets if the **offset** option is set **on** and there are no errors. **parseonly** should not be used inside a stored procedure or trigger.
- **procid** – returns the ID number of the stored procedure to Open Client DB-Library (not to the user) before sending rows generated by that stored procedure.
- **quoted_identifier** – determines whether SQL Server recognizes delimited identifiers. By default, **quoted_identifier** is set **off** and all identifiers must conform to the rules for valid identifiers. If you **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. You must enclose the identifiers within double quotation marks. Delimited identifiers cannot exceed 28 bytes, cannot be used as parameters to system procedures, cannot be used with **bcp**, and may not be recognized by all front-end products.

  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 roles granted to you are automatically enabled. Use **set role** to turn any of these options off or back on again. For example, if you have been granted the System Administrator role, you assume the identity of Database Owner within the current database. If you wish to assume your “real” user identity, execute **set role “sa_role” off**. If you are not a user in the current database, and if there is no “guest” user, you will be unable to turn off **sa_role**, because there is no server user ID for you to assume.

  The available roles are **sa_role**, **sso_role,** and **oper_role**.
- **rowcount** – causes SQL Server to stop processing the query after the specified number of rows is returned. To turn this option off so that all rows are returned, use this command:

  ```
  set rowcount 0
  ```
- **self_recursion** – determines whether or not SQL Server allows a trigger to cause itself to fire again (called self recursion). By default, SQL 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 or stored procedure 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.
- **showplan** – generates a description of the processing plan for the query and immediately processes it unless **noexec** is set. **showplan** is often used in conjunction with **statistics io**. **showplan** allows you to see whether the query uses the indexes that are in place; **statistics io** gives you an idea of the effectiveness of the indexes. (If the indexes are not being used, you may want to issue the **update statistics** command, discussed later in this chapter.) **showplan** should not be used inside a stored procedure or trigger.
- **statistics io** – displays the number of table scans, logical accesses (cache reads), and physical accesses (disk reads) for each table referenced in the statement. It also displays the number of pages written (including pages in the transaction log). If the query is being run efficiently, the number of scans should be small. If the cache is being used well, the number of disk reads should be small in relation to the number of cache reads. Turning this option on slows down execution, so it should not be used all the time.
- **statistics time** – displays the CPU time it took to parse and compile each command. It also displays the CPU time it took to execute

each step of the command. CPU times are given in units of ticks, which are machine dependent.

- **string_rtruncation** – determines if SQL Server raises an exception error when an insert or update 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 exception error and the character string is silently truncated.
- **textsize** – specifies the size in bytes of *text* or *image* type data to be returned with a **select** statement. The *@@textsize* global variable stores the current setting.
- **transaction isolation level** – causes SQL Server to automatically apply a **holdlock** to all **select** operations in a transaction when set to isolation level 3. By default, SQL Server's transaction isolation level is 1 which allows shared locks on data. You cannot change an isolation level within a transaction.

The **set** options can be divided into these categories:

- **parseonly**, **noexec**, **showplan**, **rowcount**, **textsize** and **nocount** control the way a query is executed. It doesn't make sense to set both **parseonly** and **noexec** to **on**.

  The default setting for **rowcount** is 0 (return all rows) and for **textsize** is 32768 (32K). To restore default behavior after changing these settings, set the value to 0. (Some client software sets **textsize** to other values as part of logging in to SQL Server.)

  The default for **parseonly, noexec, showplan** and **nocount** is **off**.
- The **statistics io** and **statistics time** options display performance statistics after each query. The default setting for these options is **off**.
- **arithabort** and **arithignore** determine how to handle arithmetic errors.
- **datefirst**, **dateformat**, and **language** affect date functions, date order, and message display. In the default language, us_english, **datefirst** is 1 (Sunday), **dateformat** is *mdy*, and messages are displayed in us_english. **set language** implies that SQL Server should use the first weekday and date format of the language it specifies, but it will not override an explicit **set datefirst** or **set dateformat** command issued earlier in the current session.
- Open Client DB-Library uses **offsets** and **procid** to interpret results from SQL Server. The default setting for these options is **on**.
- **char_convert** controls character set conversion between SQL Server and a client.

- **cursor rows** and **close on endtran** affect the way SQL Server handles cursors. The default setting for **cursor rows** with all cursors is 1. The default setting for **close on endtran** is **off**.
- **identity_insert** allows or prohibits inserts that affect a table's IDENTITY column.
- **chained** and **transaction isolation level** allow SQL Server to handle ANSI-compliant transactions.
- **self_recursion** allows a trigger to fire as a result of the data modification of the trigger itself. (The **sp_configure** variable **nested trigger** must also be set on.) The default setting for **self_recursion** is **off**.
- **string_rtruncation on** causes SQL Server to raise a SQLSTATE exception error when truncating a *char* or *nchar* string.
- **ansinull** on causes SQL Server to issue warning when eliminating nulls from aggregate functions and SQL equality (=) or inequality (!=) comparisons.
- **fipsflagger on** causes SQL Server to flag the use of non-ANSI SQL.
- **role** activates and deactivates roles for the current session.

If you use the **set** command inside a trigger or stored procedure, the option reverts to its former setting after the trigger or procedure executes.

All **set** options except **showplan** and **char_convert** take effect immediately. **showplan** takes effect in the following batch. Here is an example using **set show plan 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)
```

If the two statements are put in separate batches, here's what happens:

```
set showplan on
go
select * from publishers
go
```

```
STEP 1
the type of query is SELECT
FROM TABLE
publishers
nested iteration
Table Scan

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)
```

The Database Options

The system procedure **sp_dboption** changes settings for an entire database. These options remain in effect until they are changed. The **set** commands described above, which remain active only for a session, or for the duration of a stored procedure. Configuration variables, described later in this chapter, affect the entire server.

The procedure:

- Displays a list of the settable database options when it is used without a parameter
- Changes a database option when used with parameters

You can only change options for user databases, you cannot change options for the *master* database. However, to change a database option in a user database (or to display a list of the database options), **sp_dboption** must be executed while using the *master* database.

The syntax for **sp_dboption** is:

sp_dboption [*dbname*, *optname*, {true | false}]

To make an option or options take effect for every new database, change the option in the *model* database.

Listing the Database Options

All users with access to the *master* database can execute **sp_dboption** with no parameters in order to display a list of the database options. The report from **sp_dboption** looks like this:

```
sp_dboption
```

```
Settable database options.
--------------------
abort tran on log full
allow nulls by default
auto identity
dbo use only
ddl in tran
no chkpt on recovery
no free space acctg
read only
select into/bulkcopy
single user
trunc log on chkpt
trunc. log on chkpt.
```

Here's what the database options mean:

- **abort tran on log full** determines the fate of a transaction that is running when the last-chance threshold is crossed. The default value is **false**, meaning that the transaction is suspended and is awakened only when space has been freed. If you change the setting to **true**, all user queries that need to write to the transaction log are killed until space in the log has been freed.
- Setting **allow nulls by default** to **true** changes the default nulltype of a column from **not null** to **null**, in compliance with the ANSI standard. The Transact-SQL default value for a column is **not null**, meaning that null values are not allowed in a column unless **null** is specified in the column definition.
- While the **auto identity** option is **true**, a 10-digit IDENTITY column is defined in each new table that is created without specifying either a **primary** key, a **unique** constraint, or an IDENTITY column. The column is not visible when you select all column with the **select *** statement. To retrieve it, you must explicitly mention the column name, *SYB_IDENTITY_COL*, in the select list.
- While the **dbo use only** option is set on (or true), only the Database Owner can use the database.

- Setting **ddl in tran** option is to **true** allows the following commands to be used inside a user-defined transaction:

alter table	**create table**	**drop rule**
create default	**create trigger**	**drop table**
create index	**create view**	**drop trigger**
create procedure	**drop default**	**drop view**
create rule	**drop index**	**grant**
create schema	**drop procedure**	**revoke**

Table 12-1: DDL Commands Allowed in Transactions

Data definition statements must lock system tables for the duration of a transaction, and can result in performance problems.

- The following commands cannot be used inside a user-defined transaction under any circumstances:

alter database	**load database**	**truncate table**
create database	**load transaction**	**update statistics**
disk init	**reconfigure**	
drop database	**select into**	

Table 12-2: DDL Commands Not Allowed in Transactions

- **no free space acctg** suppresses free space accounting and execution of threshold actions for the non-log segments. This speeds recovery time because the free-space counts will not be recomputed for those segments. It disables updating the rows-per-page value stored for each table, so system procedures that estimate space usage may report inaccurate values.
- The **no chkpt on recovery** option is set on (true) when an up-to-date copy of a database is kept. In these situations, there is a "primary" and a "secondary" database. Initially, the primary database is dumped and loaded into the secondary database. Then, at intervals, the transaction log of the primary database is automatically dumped and loaded into the secondary database.

  If this option is set off (false)—the default condition—a checkpoint record is added to the database after it is recovered due to restarting SQL Server. This checkpoint, which ensures that the recovery mechanism won't unnecessarily be re-run, changes the sequence number on the database. If the sequence number on the secondary database has changed, a subsequent dump of the transaction log from the primary database could not be loaded into it.

Turning on this option for the secondary database causes it not to get a checkpoint from the recovery process, so that subsequent transaction log dumps from the primary database can be loaded into it.

- The **read only** option means that users can retrieve data from the database, but can't modify anything.
- The **select into/bulkcopy** option must be set on in order to perform operations that do not keep a complete record of the transaction in the log:
  - To use the **writetext** utility
  - To **select into** a permanent table
  - To do a "fast" bulk copy with **bcp.** "Fast" **bcp** is used by default on tables which do not have indexes.

  SQL Server performs minimal logging for these commands, recording only page allocations and de-allocations, but not the actual changes that are made on the data pages.

  You do not have to set the **select into/bulkcopy** option on in order to **select into** a temporary table, since *tempdb* is never recovered. The option does not need to be set in order to run **bcp** on a table that has indexes, because inserts are logged.

  After you have run a **select into** command or performed a bulk copy in a database, you won't be able to perform a regular transaction log dump. Once you have made minimally-logged changes to your database, you must perform a **dump database** since changes would not be recoverable from transaction logs.

  Just setting the **select into/bulkcopy** option doesn't block log dumping, but making minimally-logged changes to data does block the use of a regular **dump transaction.** You can still use **dump transaction...with no_log** and **dump transaction...with truncate_only,** however.

  By default, the **select into/bulkcopy** option is off in newly created databases. To change the default situation, turn this option on in the *model* database.
- When **single user** is set to TRUE, only one user at a time can access the database.
- The **trunc log on chkpt** option means that the transaction log is truncated (committed transactions are removed) every time the **checkpoint** checking process occurs (usually more than once per

minute). When the Database Owner runs **checkpoint** manually, however, the log is **not** truncated.

It may be useful to turn this option on while doing development work during which backups of the transaction log are not needed. If this option is off (the original default condition) and the transaction log is never dumped, the transaction log continues to grow and you may run out of space in your database.

When the **trunc log on chkpt** option is on, you cannot dump the transaction log because changes to your data are not recoverable from transaction log dumps. In this situation, issuing the **dump transaction** command produces an error message instructing you to use **dump database** instead.

By default, the **trunc log on chkpt** option is off in newly created databases. To change the default situation, turn this option on in the *model* database.

◆ ***WARNING!***

If you turn trunc log on chkpt on in model, and need to load a set of database and transaction logs into a newly created database, be sure to turn the option off in the new database.

For a report on which database options are set in a particular database, execute the system procedure **sp_helpdb** in that database.

Setting the Database Options

Only a System Administrator or Database Owner can change a user database's options by executing **sp_dboption**. You must be using the *master* database to execute **sp_dboption**. Then, in order for the change to take effect, you must issue the **checkpoint** command while using the database for which the option was changed.

Remember that none of the *master* database's options can be changed.

Here's how you'd use **sp_dboption** to change the *pubs2* database to **read only**:

```
use master

sp_dboption pubs2, "read only", true
```

Then run the **checkpoint** command in the database that was changed:

```
use pubs2

checkpoint
```

For the *optname* parameter, SQL Server understands any unique string that is part of the option name. To set the **trunc log on chkpt** option, you can issue this command:

```
use master

sp_dboption pubs2, trunc, true
```

If you enter a value for *optname* that is ambiguous, an error message is displayed. For example, two of the database options are **dbo use only** and **read only.** Using "only" for the *optname* parameter generates a message because it matches both names. The complete names that match the string supplied are printed out so you can see how to make the *optname* more specific.

More than one database option can be set on at a time. You cannot change database options inside a user-defined transaction.

Monitoring SQL Server Activity

SQL Server keeps track of how much work it has done in a series of pre-defined global variables, distinguished from local variables by having two @ signs preceding their names, for example, *@@error*, *@@rowcount*.

Executing **sp_monitor** displays the current values of some of these global variables, and how much they have changed since the last time the System Administrator ran the procedure. You can also query the global variables directly.

The **sp_monitor** procedure takes no parameters. Here's how you use it, and a sample report from it:

```
sp_monitor
```

```
last_run            current_run         seconds
------------------- ------------------  -------
Jan 29 1993 10:11AM Jan 29 1993 10:17AM 314

cpu_busy         io_busy   idle
---------------- --------- ---------------
4250(215)-68%    67(1)-0%  109(100)-31%

packets_received packets_sent packet_errors
---------------- ------------ ------------
781(15)          10110(9596)  0(0)

total_read   total_write total_errors  connections
----------   ----------- ------------  -----------
394(67)      5392(53)    0(0)          15(1)
```

For each column, the statistic is printed in the form *number(number)* or *number(number)-number%*.

The first number refers to the number of seconds (for *cpu_busy*, *io_busy*, and *idle*) or the total number (for the other variables) since SQL Server was restarted. The number in parentheses refers to the number of seconds or total number since the last time **sp_monitor** was run. The percentage is the percent of time since **sp_monitor** was last run.

For example, this report shows *cpu_busy* as 4250(215)-68%. This means that the CPU has been busy 4250 seconds since SQL Server was last started up and 215 seconds since **sp_monitor** was last run. The 68% means that the CPU has been busy 68% of the time since **sp_monitor** was last run.

For the *total_read* variable, the value 394(67) means there have been 394 disk reads since SQL Server was restarted, 67 of them since the last time **sp_monitor** was run.

Here are the columns in the **sp_monitor** report, the equivalent global variables, if any, and their meanings:

Column Name	Variable	Meaning
last_run		The clock time at which the **sp_monitor** procedure was last run.
current_run		The current clock time.
seconds		The number of seconds since **sp_monitor** was last run.

Table 12-3: Meaning of Columns in an sp_monitor Report

Column Name	Variable	Meaning
cpu_busy	*@@cpu_busy*	The number of seconds in CPU time that SQL Server's CPU was doing SQL Server work.
io_busy	*@@io_busy*	The number of seconds in CPU time that SQL Server has spent doing input and output operations.
idle	*@@idle*	The number of seconds in CPU time that SQL Server has been idle.
packets_received	*@@pack_received*	The number of input packets read by SQL Server.
packets_sent	*@@pack_sent*	The number of output packets written by SQL Server.
packet_errors	*@@packet_errors*	The number of errors detected by SQL Server while reading and writing packets.
total_read	*@@total_read*	The number of disk reads by SQL Server.
total_write	*@@total_write*	The number of disk writes by SQL Server.
total_errors	*@@total_errors*	The number of errors detected by SQL Server while reading and writing.
connections	*@@connections*	The number of logins or attempted logins to SQL Server.

Table 12-3: Meaning of Columns in an sp_monitor Report (continued)

The global variables report time in machine ticks, while **sp_monitor** reports seconds of CPU time. The global variable *@@timeticks* reports the number of microseconds per tick on your platform.

Here's how to query a global variable directly:

```
select @@cpu_busy

----------------
           42506
```

There are some other global variables, whose values are not reported by **sp_monitor**. They are:

- *@@char_convert* – contains 0 if character set conversion not in effect. Contains 1 if character set conversion is in effect.
- *@@client_csname* – client's character set name. Set to NULL if client character set has never been initialized; otherwise, it contains the name of the most recently used character set.

- *@@client_csid* – client's character set id. Set to NULL if client character set has never been initialized. Otherwise, it contains the most recently used client character set's *id* from *syscharsets*.
- *@@error* – contains the last error number generated by the system. The *@@error* global variable is commonly used to check the error status (succeeded or failed) of the most recently executed statement. A statement such as:

  ```
  if @@error != 0 return
  ```

  causes an exit if an error occurs.
- *@@identity* - last value inserted into an IDENTITY column by an **insert** or **select into** statement.
- *@@isolation* - current isolation level of the Transact-SQL program. *@@isolation* takes the value of the active level (1 or 3).
- *@@langid* – defines the local language id of the language currently in use (specified in *syslanguages.langid*).
- *@@language* – defines the name of the language currently in use (specified in *syslanguages.name*).
- *@@maxcharlen* – maximum length of multibyte character in SQL Server's character set.
- *@@max_connections* – the maximum number of simultaneous connections that can be made with SQL Server in this computer environment. The user can configure SQL Server for fewer connections with **sp_configure**.
- *@@ncharsize* – average length of a national character.
- *@@nestlevel* – nesting level of current execution (initially zero). Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. If the maximum of 16 is exceeded, the transaction aborts.
- *@@procid* – contains the stored procedure ID of the currently executing procedure.
- *@@rowcount* – contains the number of rows affected by the last command.
- *@@servername* – the name of this SQL Server. To define a server name, a System Security Officer must run **sp_addserver** and then re-start the server.
- *@@spid* - server process ID number of the current process.
- *@@sqlstatus* - contains status information resulting from the last **fetch** statement.

- *@@textsize* – the maximum size in bytes of *text* or *image* data to be returned with a **select** statement. The default value depends on the client software, the **isql** default is 32K bytes. Can be changed on a per-session basis with **set textsize**.
- *@@thresh_hysteresis* - change in free space required to activate a threshold.
- *@@timeticks* – the number of microseconds per tick.
- *@@tranchained* - current transaction mode of the Transact-SQL program.
- *@@trancount* – contains the number of currently active transactions for the current user.
- *@@transtate* - current state of a transaction after a statement executes.
- *@@version* – contains the version string of the current version of SQL Server.

Any user can query the global variables. However, only System Administrators can execute **sp_monitor**. The procedure updates a table in the *master* database called *spt_monitor*.

update statistics

The **update statistics** command helps SQL Server make the best decisions about which indexes to use when it processes a query, by providing information about the distribution of the key values in the indexes. **update statistics** is automatically run when you create or recreate an index on a table that already contains data. It can be used when a large amount of data in an indexed column has been added, changed, or deleted. The crucial element is that the optimization of your queries depends on the accuracy of the distribution steps, and if there is significant change in the key values in your index, you should re-run **update statistics** on that index.

Only the table owner or System Administrator can issue the **update statistics** command.

Its syntax is:

```
update statistics table_name [index_name]
```

If you do not specify an index name, the command updates the distribution statistics for all the indexes in the specified table. Giving an index name updates statistics for that index only.

Try to run **update statistics** at a time when the tables you need to specify are not heavily used. **update statistics** acquires locks on the tables and indexes as it reads the data.

You can find the names of indexes by using the **sp_helpindex** system procedure. This procedure takes a table name as a parameter.

To list the indexes for the *authors* table, and then update the statistics for all of the indexes in the table, type:

```
sp_helpindex authors

update statistics authors
```

To update the statistics only for the index on the *au_id* column, type:

```
update statistics authors auidind
```

Since Transact-SQL does not require index names to be unique in a database, you must give the name of the table with which the index is associated.

After running **update statistics**, run **sp_recompile** on the table so that triggers and procedures that use the indexes will use the new distribution:

```
sp_recompile authors
```

Resetting the Configuration Variables

The configuration variables control various aspects of SQL Server's memory allocation and performance. The System Administrator or System Security Officer can reset these configuration variables in order to fine-tune performance and refine storage allocation. In the absence of intervention by the System Administrator, SQL Server supplies default values for all these variables that are reasonable given the following assumptions:

- At least 10 megabytes of RAM dedicated to SQL Server
- An application in which there is a great deal of update activity
- An application in which a relatively small number of stored procedures are used repeatedly

If your application differs significantly from these assumptions, you should reset the configuration variables immediately after installation. System variables are set by:

- Executing the system procedure **sp_configure**, which updates the *values* field of the system table *master..sysconfigures*.

- Issuing the **reconfigure** command. Note that the **allow updates** configuration variable requires **reconfigure with override**, since this option allows you to edit system tables directly.
- Restarting SQL Server for most of the configuration variables. Only a few of the configuration variables are dynamic, meaning that they can be changed using only **sp_configure** and **reconfigure**, and do not require a server reboot.

The *sysconfigures* and *syscurconfigs* Tables

The *master..sysconfigures* and *master..syscurconfigs* system tables store configuration variables. The *value* field contains the value that has been set by executing **sp_configure** (though it does not reflect the value SQL Server is using if **reconfigure** was not run).

The *status* field of *sysconfigures* cannot be updated by the user. Status 1 means "dynamic," indicating that new values for these configuration variables take effect immediately when the **reconfigure** command is issued. The rest of the configuration variables—those with status "0"—take effect only after the **reconfigure** command is issued **and** SQL Server is next restarted.

The *sysconfigures* table does not reflect the values currently being used by SQL Server, for two reasons:

- The System Administrator may have updated *sysconfigures* by issuing **sp_configure** commands, but not issued the **reconfigure** command and / or not restarted SQL Server, or
- The default for many values in *sysconfigures* is 0, which indicates that SQL Server calculates default values for that variable.

The run values are stored in the "phantom" system table *syscurconfigs*. This table is created dynamically when it is queried.The system procedure **sp_configure** can perform a join on *sysconfigures* and *syscurconfigs* to display the values that are currently in use—the **run values**—and the configured values.

Changing the Configuration Variables: *sp_configure* and *reconfigure*

The system procedure **sp_configure** displays and resets configuration values. You can use it in these ways:

- If you use it without any arguments, it displays all the configuration variables, their minimum and maximum values, the configuration value, and the "run value", the value that SQL Server is currently using.

- If you give a configuration variable name, it prints the values for that variable. You do not have to type the entire name, you only have to type enough characters to make your input unique.
- If you give a non-unique fragment of a variable name, it prints a list of all of the variable names that match the fragment.
- If you give a variable name and a value, it resets the configuration value of that option in the system tables. For nearly all of the configuration options, you must run **reconfigure** and restart the server to make the new values to take effect.

The **sp_configure** syntax is:

sp_configure [*optname* [, *optvalue*]]

Any user can execute **sp_configure** with the first parameter (*optname*) or with no parameters. System Administrators can execute **sp_configure** with both parameters, except for the **password expiration interval, audit queue size, allow updates**, and **remote access** variables, for which you must be a System Security Officer.

The sample output below shows the kind of information that **sp_configure** prints when you execute it with no parameters. The values that it prints will vary, depending on your platform and on what values you have already changed.

Configuration Variable	Minimum	Maximum	Config. Value	Run Value
recovery interval[1]	1	32767	0	5
allow updates[1]	0	1	0	0
user connections[2]	5	2147483647	0	25
memory[3]	varies	2147483647	0	varies
open databases	5	2147483647	0	12
locks	5000	2147483647	0	5000
open objects	100	2147483647	0	500
procedure cache	1	99	0	20
fill factor	0	100	0	0

1. Dynamic options, reset by running **reconfigure**. All other options require **reconfigure**, plus shutting the server down and restarting it.
2. The actual maximum value of user connections is stored in *@@max_connectio*ns. The maximum number of network connections and open devices available to SQL Server vary according to platform and operating system.
3. The *minimum* and *run_value* values for this option vary according to platform.

Table 12-4: sp_configure Output

Configuration Variable	Minimum	Maximum	Config. Value	Run Value
time slice	50	1000	0	100
database size	2	10000	0	2
tape retention	0	365	0	0
recovery flags	0	1	0	0
nested triggers[1]	0	1	1	1
devices	4	256	0	10
remote access[1]	0	1	1	1
remote logins	0	2147483647	0	20
remote sites	0	2147483647	0	10
remote connections	0	2147483647	0	20
pre-read packets	0	2147483647	0	3
upgrade version[1]	0	2147483647	1001	1001
default sortorder id	0	255	50	50
default language[1]	0	2147483647	0	0
language in cache	3	100	3	3
max online engines	1	32	1	1
min online engines	1	32	1	1
engine adjust interval	1	32	0	0
cpu flush[1]	1	2147483647	200	200
i/o flush[1]	1	2147483647	1000	1000
default character set id	0	255	1	1
stack size	20480	2147483647	0	28672
password expiration interval[1]	0	32767	0	0
audit queue size	1	65535	100	100
additional netmem	0	2147483647	0	0
default network packet size	512	524288	0	512
maximum network packet size	512	524288	0	512
extent i/o buffers	0	2147483647	0	0
identity burning set factor	1	9999999	5000	5000

1. Dynamic options, reset by running **reconfigure**. All other options require **reconfigure**, plus shutting the server down and restarting it.
2. The actual maximum value of user connections is stored in *@@max_connectio*ns. The maximum number of network connections and open devices available to SQL Server vary according to platform and operating system.
3. The *minimum* and *run_value* values for this option vary according to platform.

Table 12-4: sp_configure Output (continued)

The *config_value* column of the report contains the value to which the configuration variable has been set with **sp_configure**. It changes after you execute **sp_configure**. This is the value in *sysconfigures.value*.

The *run_value* column contains the value SQL Server is using. It changes after you run the **reconfigure** command and restart the server. This is the value in *syscurconfigs.value*.

The report displayed by **sp_configure** is partially constructed from a table called *spt_values*, which is a "work table" in the *master* database that stores records that refer to locks, permissions, configuration variables, and other system information.

To change the current values of the configuration variables, execute **sp_configure** with its parameters. The *optname* parameter is the name of the configuration variable, as given in the name column of the report from **sp_configure**. You can use the first few characters of the name, as long as they provide a unique pattern. The *optvalue* is its new value, which must be equal to or greater than the minimum and less than or equal to maximum given in the report from **sp_configure**. For the options that are on-off toggles (**allow updates**, **recovery flags**, **nested triggers** and **remote access**) the *optvalue* is either "1" for "yes", or "0" for "no".

Here is how you would decrease the number of user connections (perhaps in order to save memory):

```
sp_configure "user conn", 20
```

After executing **sp_configure**, run the **reconfigure** command.

The *reconfigure* Command

The syntax of **reconfigure** is:

```
reconfigure [with override]
```

Only a System Administrator can execute the **reconfigure** command.

When the System Administrator issues the **reconfigure** command, SQL Server checks to make sure that the new values in *sysconfigures* are acceptable and reasonable.

User-supplied values that seem unreasonable to SQL Server cause it to abort the **reconfigure** command and display a warning message. The System Administrator can then re-issue **sp_configure** with a new value, or re-issue the **reconfigure** command with the **with override** clause, which causes SQL Server to accept whatever values the System Administrator supplies.

➤ ***Note***

The **reconfigure** command performs certain simple sanity checks on configuration values. It is possible, however, for you to allocate so much memory to SQL Server that it cannot be rebooted, or allocate so much memory for certain memory uses that there is little or no room left in the server for page and procedure buffers. You can always restore SQL Server's default configuration variables, and execute your configuration commands again.

One of the configuration variables, **allow updates**, always requires **reconfigure with override**, and can be set only by a System Security Officer.This requirement is meant to supply a small measure of added protection—to cause you to consider the consequences of allowing direct updates to the system tables.

For certain values set to zero in *sysconfigures*, SQL Server calculates a default. For example, if the user gives 0 for the number of **open databases**, SQL Server might substitute the value 12. Whenever **remote access** is set to 1, and the other configuration variables that affect remote access are set to 0, SQL Server calculates default values for each of them. (These are listed in Chapter 15, ‘‘Managing Remote Servers''.)

When the **reconfigure** command is accepted, SQL Server writes the new configuration variables to the area of the disk that holds the configuration structure. If you changed the value of a dynamic variable, **reconfigure** updates the run value, and the change takes effect immediately. If you changed a non-dynamic variable, the change takes effect until the next time SQL Server is restarted.

Details on Configuration Variables

The following sections give additional information on the configuration variables.

recovery interval

The **recovery_interval** sets the maximum number of minutes per database that SQL Server should use to complete its recovery procedures in case of a system failure.

SQL Server uses this number and the amount of activity on each database to decide when to checkpoint each database. When SQL Server checkpoints a database, it writes all dirty pages (data pages which have been changed by data modification commands) to the disk. The checkpoint also performs a few other "housekeeping" tasks, including truncating the log if this option has been set with **sp_dboption**. About once per minute, the sleeping checkpoint process "wakes up", checks the **truncate log on checkpoint** setting, and checks the recovery interval to determine if it is time to perform a checkpoint. The following illustration shows the logic used by SQL Server during this process.

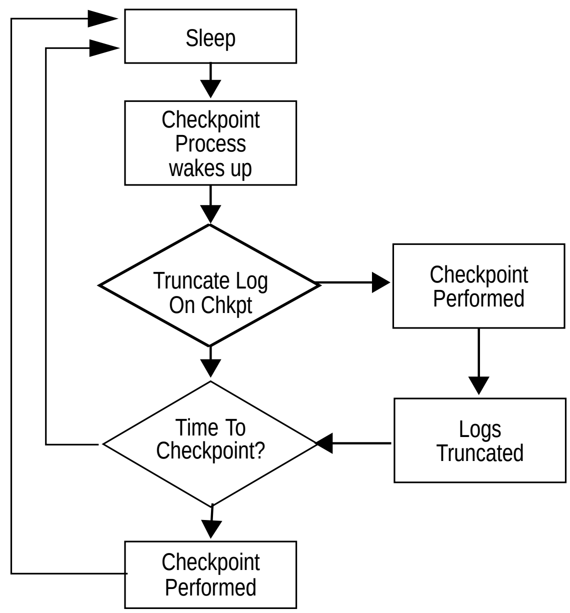


Figure 12-1: The Checkpoint Process

You may want to change the recovery interval if your application and its usage change. For example, you may want to shorten the recovery interval when there is an increase in update activity on SQL Server. Shortening the recovery interval causes more frequent checkpoints, which slows the system very slightly. (A typical checkpoint takes about one second.) On the other hand, setting the recovery interval too high might cause the time to recover to be unacceptably long.

The default run value for this variable is 5 (minutes per database).

This variable is dynamic, which means that an updated value takes effect as soon as the System Administrator issues the **reconfigure** command.

➤ ***Note***

If failure occurs during a long-running transaction, recovery time may be longer than the specified recovery interval, since SQL Server must roll back the changes during recovery.

allow updates

By default, system tables can be updated only indirectly, using system procedures. The **allow updates** variable allows you to change this situation. This variable can be set only by System Security Officers.

The **allow updates** variable is a toggle: it is either on or off. When the value is changed from the default 0 (off) to 1 (on), any user with appropriate permissions can update the system tables directly with ad hoc queries, and can create stored procedures that update the system tables.

◆ ***WARNING!***

Updating certain fields in the system tables causes SQL Server to be unable to run. Therefore, allowing direct updates to the system tables is risky. When you update system tables, always include the data modification statements in a user-defined transaction so that you can roll back the changes if necessary. Check the results carefully before you commit the transaction.

Stored procedures that you create while the **allow updates** variable is set on are always able to update the system tables, even after the variable has been set back to off. Whenever you turn the **allow updates** toggle on, you are creating a "window of vulnerability"—a period of time during which SQL Server users can alter the system tables or create a stored procedure with which the system tables can be altered in the future.

Because the system tables are so critical, it is best to turn this toggle on only in highly controlled situations. If you want to guarantee that no other users can access SQL Server while the system tables can be directly updated, you can restart SQL Server in single user mode. For details, see the man pages on **startserver** and **dataserver** in the *SQL Server Utility Programs* manual for your operating system.

allow updates is a dynamic option, which means that a new value takes effect as soon as the System Administrator issues the **reconfigure with**

override command. The **with override** clause is always required for this configuration variable as an added measure of protection.

user connections

This variable sets the maximum number of user connections that can be connected to SQL Server at the same time. It does not refer to the maximum number of processes; that number depends not only on the value of this variable but also on other system activity.

The maximum allowable number of connections (or file descriptors) is operating system dependent; see the *System Administration Guide Supplement* for your platform.

The number available for SQL Server is stored in the global variable *@@max_connections*. You can get a report on the maximum number of file descriptors that your system can use with this statement:

```
select @@max_connections
```

This number represents the maximum number of file descriptors allowed by the system for your process, minus the following file descriptors used by SQL Server:

- One for each configured master network listener (one for every "master" line in the interfaces file entry for that SQL Server)
- One for standard output
- One for the error log file
- One for internal use (VMS only)

This value does not include the count of or the number of site handlers active at a given time. (For SQL Servers running on OpenVMS, this value also subtracts one connection for a debug port.)

◆ ***WARNING!***

If user connections is set higher than the result of the computation described below, some SQL Server processes, such as remote procedure calls or periodic dumps to tape or disk files, may fail. Sybase Technical Support can help you reset the maximum number of connections for your configuration.

SQL Server allocates approximately 51K of memory as overhead per user connection. If you increase the **stack size** or **default packet size** configuration variables, the amount of memory per user connection increases also.

In addition, you must reserve a number of connections for:

- The data devices, including the master device
- Backup Server
- Mirror devices
- The maximum number of site handlers that are active at any given time

To determine these values, use these three queries:

```
select count(*) from master..sysdevices
where cntrltype = 0

select count(*) from sysdevices
where mirrorname is not NULL

select count(*) from sysservers
where srvname != @@servername
```

The maximum value to which **user connections** can be set is the value of *@@max_connections* minus the sum of the results of these three queries.

There is no formula for determining how many connections to allow for each user. Rather, you must estimate this number based on the system and user requirements outlined here. You must also take into account that on a system with many users, there is more likelihood that connections needed only occasionally or transiently can be shared among users.

- One connection is needed for each user running **isql.**
- Application developers use one connection for each editing session.
- The number of connections required by users running an application depends entirely on how the application has been programmed. Users executing Open Client programs need one connection for each open DB-Library **dbprocess** or Client-Library **cs_connection**.

➤ **Note**

It's a good idea to estimate the maximum number of connections that will be used by SQL Server and to update **user connections** as you add physical devices or users to the system. Use **sp_who** periodically to determine the number of users on your SQL Server.

memory

This variable sets the size of memory, in 2K units, that SQL Server allocates from the operating system. The default value of **memory** varies from platform to platform: see the *SYBASE SQL Server Installation Guide* for its value on your operating system.

The amount of memory configured must be enough for:

- SQL Server's static memory needs (kernel overhead, user stack space, etc.)
- The procedure cache and the data cache (also called buffer cache). Changing caches memory allocations is discussed later.
- Network buffers for each user connection allocated with **default network packet size**, discussed later.
- Buffers used for **extent i/o buffers**, discusses later.

The more memory available, the more resources SQL Server has for internal buffers and caches—reducing the number of times the server has to read data from disk for static information or compiled procedure plans. There is no performance penalty for configuring SQL Server to use the maximum memory available to it on your computer. However, you must be sure to assess other memory needs on your system, or SQL Server may not be able to acquire enough memory to boot.

To maximize the amount of **memory** for SQL Server on your system:

- Subtract the memory required for the operating system from the total physical memory on your computer system.
- If the machine is not wholly dedicated to SQL Server, subtract the memory requirements for other system uses also. (Windowing systems, such as X Windows, require a lot of memory and can interfere with SQL Server performance when used on the same machine as SQL Server.)
- Subtract memory allocated for **additional netmem**. This is explained in "additional netmem" on page 12-38.

The memory left over after SQL Server's static memory needs are met is divided between the procedure cache and the data cache. The percentage allocated to the procedure cache is set by the **procedure cache** configuration variable.

Change the value of the **memory** configuration variable:

- When you add or remove memory on the computer system
- When the use of the computer system changes
- If you allocate memory for extent I/O buffers or additional network memory for SQL Server

You can also increase the value to enhance the performance of SQL Server when applications need to use a lot of memory (for example, when using large indexes).

If SQL Server Cannot Boot

When SQL Server is booted, it acquires as much of the specified memory as the system allows. If this amount of memory is not available, SQL Server acquires as much as it can.[1] If the configured size is extremely large relative to the available memory, SQL Server cannot boot because it cannot allocate the requested amount of memory.

If SQL Server does not boot for this reason, you must run **buildmaster** with the "reconfigure" command line option to reset the configuration block to default values. This resets the *run_value* of all configuration variables to their default values. The *config_value* retains the settings you have set. You must reset **memory** value or other configuration values that affect memory use, and then run **reconfigure** and re-boot the server to install the new values that you have changed. See the *SQL Server Utility Programs* manual for more information about **buildmaster**.

open databases

This variable sets the maximum number of databases that can be open at one time on SQL Server. The system databases *master*, *model*, *sybsystemprocs* and *tempdb* are included in the number of open databases. If you have installed auditing, the *sybsecurity* database must be counted. The sample database *pubs2* and the syntax database *sybsyntax* are optional databases, but you must also count them if they are installed.

1. If your operating system supports dynamic memory allocation throughout the life of a process, it may allocate additional memory to SQL Server after it has started.

The default run value is 12. Setting the number of open databases higher does not have a significant impact on performance or storage requirements. Therefore, don't hesitate to increase this value if you are adding new user databases to your server, or if SQL Server ever displays a message saying that you've exceeded the allowable number of open databases. Each open database requires approximately 17K of memory.

locks

This variable sets the number of available locks for all users on SQL Server. Locks are not shared the way open databases and database objects are.

The default run value in *syscurconfigures* is 5000. If this number proves insufficient, SQL Server displays a message.

Setting the number of available locks higher does not have a significant impact on performance or storage requirements. Therefore, don't hesitate to increase this value if SQL Server ever displays a message saying that you've exceeded the allowable number of available locks.

open objects

This variable sets the maximum number of database objects that can be open at one time on SQL Server.

The default run value is 500. If this number proves insufficient, SQL Server displays a message.

Setting the number of open database objects higher does not have a significant impact on performance or storage requirements. Therefore, don't hesitate to increase this value if SQL Server ever displays a message saying that you've exceeded the allowable number of open database objects.

procedure cache

This variable gives the percentage of memory allocated to the procedure cache after SQL Server's memory needs are met. SQL Server's memory needs are the sum of memory necessary for locks, user connections, the code itself (which varies slightly from release to release), and other resources. The remaining memory is divided between the procedure cache and the data cache according to the percentage given by this configuration variable.

SQL Server stores compiled stored procedures in the area of memory called the procedure cache. If the server finds a procedure or a compilation already in the cache, it doesn't need to read it from the disk. SQL Server also uses space in the procedure cache to compile queries and while creating stored procedures.

SQL Server loads data pages and index pages into the area of memory called the data cache. If SQL Server finds a data page or index page that has already been called by a user in the cache, it isn't necessary to read it from the disk. The data cache holds most recently-used data and index pages; older pages are copied to disk or flushed as space is needed for new pages.

Both caches are managed in a LRU-MRU (least recently used, most recently used) fashion.

The default run value for **procedure cache** is 20, which gives the procedure cache 20% of memory that remains after SQL Server's requirements are met. By default, the data cache gets the other 80%.

Since the optimum value for this configuration variable is different from application to application, resetting it may improve SQL Server's performance. For example, if you run many different procedures or ad hoc queries, your application will use the procedure cache more heavily, so you may want to increase this value.

Many applications fall in this category during development. You may want to try setting this variable to 50 during your development cycle, and resetting it to 20 when your application becomes stable.

fillfactor

This variable determines how full SQL Server makes each page when it is creating a new index on existing data (unless the user specifies some other value in the **create index** statement). The **fillfactor** percentage affects performance, because SQL Server must take the time to split pages when they fill up. There is seldom a reason to change the **fillfactor** variable, especially since you can override it in the **create index** command.

The **fillfactor** percentage is used only at the time the index is created, and becomes less important as more changes are made to the data. The pages are not maintained at any particular level of fullness.

The default **fillfactor** run value is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless it has been changed with **sp_configure**). Legal values when specifying a **fillfactor** are 1 to 100.

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

A **fillfactor** of 0 does not mean that pages are 0% full. Rather, it is treated like a **fillfactor** of 100 in that SQL Server creates clustered indexes with completely full data pages, and nonclustered indexes with completely full leaf pages. It is different from 100 in that SQL Server leaves a comfortable amount of space within the index B-tree in both cases.

Other **fillfactor** values cause SQL Server to create new indexes with pages that are not completely full. For example, a **fillfactor** of 10 might be a reasonable choice if you are creating an index on a table that you know contains a small portion of the data it will eventually hold. Smaller **fillfactor** values cause each index to take more storage space.

time slice

This variable sets the number of milliseconds that a user process is allowed to run by SQL Server's scheduler (which is invisible to the user). If **time slice** is set too low, SQL Server may spend too much time switching processes. If it is set too high, users may experience lengthy response time.

The default run value is 100 milliseconds. There is seldom reason to change it.

database size

This variable sets the default number of megabytes allocated to a new user database if the **create database** statement is issued without any size parameters. A database size given in a **create database** statement takes precedence over the value set by this configuration variable.

The default run value is 2 (megabytes). If most of the new databases on your SQL Server require more than 2 megabytes, you may want to increase this value.

➤ ***Note***

You must increase the **database size** value if your *model* database grows to be larger than 2 megabytes, since during execution of the **create database** statement SQL Server copies *model* to create a new user database.

tape retention

This variable can be used to set the number of days that you expect to retain each backup tape after it has been used for a database or transaction log dump. If you try to use the tape before that many days have passed, Backup Server issues a warning message to the tape operator. You can override the warning if you wish.

The default run value is 0. Unless you change it, no warning is issued. A typical value might be 7 (days).

Both the **dump database** and **dump transaction** commands provide a **retaindays** option, which overrides the **tape retention** value for a particular dump. See “Protecting Dump Files from Being Overwritten” on page 8-20 for more information.

recovery flags

This variable sets a toggle that determines what information SQL Server displays on the console during recovery. The default run value is 0, which means that SQL Server displays only the database name and a message saying that recovery is in progress. The other legal value is 1, which means that SQL Server displays information about each individual transaction, including whether it was aborted or committed.

nested triggers

This option is a toggle that controls the use of nested triggers. When the option is 1, data modifications made by triggers can fire other triggers. A **set** option, **self_recursion**, controls whether triggers re-fire on data modifications made Set the option to 0 to disable nested triggers. The default is 1.

devices

This variable controls the number of database devices that SQL Server can use. It does not include devices used for database dumps.

If you want to lower the **devices** value after you have added database devices, you must first check to see what devices numbers are already in use by database devices. The following command prints these highest value in use:

```
select max(low/power(2,24))
    from master..sysdevices
```

The master device is device 0, so you must add one to the number that this query returns to find the minimum value you can use to reconfigure the **devices** variable. If you set this value too low, SQL Server will not be able to recover databases that use devices with higher numbers.

The default value is 10. Each device uses less than 1/2 K of memory.

remote access

This variable is a toggle that controls logins from remote SQL Servers. The default value is 1, so that SQL Server can communicate with Backup Server. Only a System Security Officer can set **remote access**.

Setting the value to 0 disables server-to-server remote procedure calls. Since SQL Server communicates with its Backup Server via remote procedure calls, setting this variable to 0 makes it impossible to back up a database. Since other system administration actions are required to enable servers besides Backup Server to execute remote procedure calls, leaving this option set to 1 does not comprise a security risk.

The next four options listed by **sp_configure**, **remote logins**, **remote sites**, **remote connections** and **pre-read packets** are set to default values whenever the **remote access** option is 1. See the complete discussion of these options in Chapter 15, "Managing Remote Servers".

upgrade version

The upgrade program provided with new releases changes this number. You won't need to change it with **sp_configure**; it merely provides a way to check SQL Server to see that the latest upgrade utility has been run.

default sortorder id

This is the number of the sort order that is currently installed as the default on the server. If you want to change the default sort order, see Chapter 17, "Language, Sort Order, and Character Set Issues".

default language

This is the number of the language that is used to display system messages unless a user has chosen another language from those available on the server. us_english always has an ID of 0. Additional languages are assigned unique numbers as they are added.

language in cache

This number indicates the maximum number of languages that can be simultaneously held in the language cache. The default is 3.

max online engines

This number controls the number of engines in symmetric multiprocessor environment. The default is 1. See Chapter 14, "Managing Multiprocessor Servers", for a detailed discussion of how to set this variable for your SMP environment.

The next options, **min online engines** and **engine adjust interval**, are not currently used.

cpu flush

This number is used in chargeback accounting. It specifies how many machine clock ticks to accumulate before adding usage statistics to *syslogins* for each user. See "Configuration Variables for Chargeback Accounting" on page 4-23 for more information.

i/o flush

This number is used in chargeback accounting. It specifies how many read or write I/Os to accumulate before flushing the data to *syslogins*. See "Configuration Variables for Chargeback Accounting" on page 4-23 for more information.

default character set id

This is the number of the default character set used by the server. The default is set at installation time, and can be changed with **sybinit**. See Chapter 17, "Language, Sort Order, and Character Set Issues" for a discussion of how to change character sets and sort orders.

stack size

Certain queries, particularly those with dozens or hundreds of clauses in a **where** *column_name* **in** clause, can exceed the configured capacity of the server's execution or argument stack. Whenever these queries, or other queries that require large amounts of stack space, are run, SQL Server prints an error message and rolls back the transaction. The two options are to break these large queries into smaller queries, or to increase the size of the server's stack. Changing the stack size affects the amount of memory required for each **user connection**. The following command sets the stack size to 30K:

```
sp_configure "stack size", 30720
```

When you run **reconfigure**, the server checks the value to determine whether it is an even multiple of your SQL Server's page size. If the value is not an even multiple, **reconfigure** rounds the value up to the nearest page-size multiple, and print an informational message indicating the value.

password expiration interval

This is the number of days that passwords remain in effect after they are changed. If **password expiration interval** is set to 0 (the default value), passwords do not expire. If it is set to a number greater than 0, all passwords expire after the specified number of days. An account's password is considered expired if an interval greater than *number_of_days* has passed since the last time the password for that account was changed.

When the number of days remaining before expiration is less than 25% of the value of **password expiration interval** or seven days, whichever is greater, each time the user logs in he or she is greeted with a message giving the number of days remaining before expiration. Of course, users can change their passwords any time before expiration.

When an account's password has expired, the user can still log into SQL Server but cannot execute any commands until the user has used **sp_password** to change his or her password. If the user issues any command other than **sp_password**, the user receives an error message and the command fails. If the System Security Officer changes the user's password while the account is in **sp_password**-only mode, the account returns to normal once the new password is assigned.

This applies only to login sessions established after the password has expired. Users who are logged in at the time that their passwords expire are not affected until the next time they log in.

The default for **password expiration interval** is 0 (no expiration). This variable can be set only by System Security Officers.

audit queue size

The in-memory audit queue holds audit records generated by user processes until the records can be processed and written to the audit trail. A System Security Officer can change the size of the audit queue with the **audit queue size** configuration variable. There is a performance / risk trade-off that must be considered when you configure the queue size. If the queue is too large, records can remain in it for some time. As long as records are in the queue, they are at risk of being lost if the system crashes. However, if the queue is too small it can repeatedly become full, which effects overall system performance: user processes that generate audit records sleep if the audit queue is full.

The default value is 100, meaning that the audit queue can store up to 100 records. The minimum value is 1, the maximum is 65535.

Here are some guidelines for determining how big your audit queue should be. You must also take into account the amount of auditing to be done at your site.

- The memory requirement for a single audit record is 424 bytes.
- The maximum number of audit records that can be lost in a system crash is the size of the audit queue (in records), plus 20. After records leave the audit queue they remain on a buffer page until they are written to *sysaudits*, on the disk. The *sysaudits* pages are flushed to disk at most every 20 records (less if the audit process is not constantly busy).
- Although the memory requirement for a single audit record is 424 bytes, a record can be as small as 22 bytes when it is written to a data page.
- In *sysaudits*, the *extrainfo* field and fields containing names are of variable length, so audit records that contain full name information are generally larger.

Thus, the number of audit records that can fit on a page varies roughly from 4 to as many as 80 or more. The memory requirement for the default audit queue size of 100 is approximately 42K.

additional netmem

additional netmem sets the maximum size of additional memory that can be used for network packets that are larger than SQL Server's default packet size. The default value for **additional netmem** is 0, which means that no extra space has been allocated for large packets. See the discussion below, under **maximum network packet size**, for information on setting this configuration variable.

Memory allocated with **additional netmem** is added to the memory allocated by **memory.** It does not affect other SQL Server memory uses.

default network packet size

This variable configures the default packet size for all SQL Server users. The default value is 512 bytes. You can set **default network packet**

size to any multiple of 512 bytes, up to a maximum of 524,288. Values that are not even multiples of 512 are rounded down.

Memory for all users who log in with the default packet size is allocated from SQL Server's memory pool, set with the **memory** configuration variable. This memory is allocated for network packets when SQL Server boots.

Each SQL Server user connection uses:

- 1 read buffer
- 1 read overflow buffer
- 1 write buffer

Each of these buffers requires **default network packet size** bytes. The total amount of memory allocated for network packets is:

user connections * 3 * **default network packet size**

When you provide a new value for this variable, SQL Server requires that amount of memory for each of the three buffers, for each user connection. For example, if you set the **default network packet size** to 1024 bytes, and have 50 user connections, the amount of network memory required is:

50 * 3 * 1024 = 153600 bytes

If you increase **default network packet size**, check the **memory** configuration and **user connections** variables to be sure that you leave enough space for other SQL Server memory needs. When you reboot the Server after changing the **default network packet size**, check the messages which report buffer allocation to be sure that you have enough memory remaining.

If you increase the **default network packet size**, you must also increase the **maximum network packet size** to at least the same size. Execute both **sp_configure** commands, then run **reconfigure**.

maximum network packet size

If some of your applications send or receive large amounts of data across the network, these applications can achieve significant performance improvement by using larger packet sizes. Two examples are large bulk copy operations and applications reading or writing large *text* or *image* values. Generally, you want to keep the value of **default network packet size** small for users performing short queries, and allow users who send or receive large volumes of data to request larger packet sizes by setting the **maximum network packet size** configuration variable.

The default value for **maximum network packet size** is 512 bytes. It must always be as large as, or larger than the **default network packet size**. You can reset the maximum size without resetting **default network packet size**. Values that are not even multiples of 512 are rounded down. The maximum value is 524,288 bytes.

SQL Server guarantees that every user connection will be able to log in at the default packet size. If you increase **maximum network packet size** and **additional netmem** remains set to 0, clients cannot use packet sizes that are larger than the default size: all allocated network memory will be reserved for users at the default size. In this situation, users who request a large packet size when they log in receive a warning message telling them that their application will use the default size.

To determine the value for **additional netmem** if your applications use larger packet sizes:

- Estimate the number of simultaneous users who will request the large packet sizes, and the sizes their applications will request.
- Multiply this sum by three, since each connection needs three buffers.
- Add 2% for overhead, rounded up to the next multiple of 512.

For example, if you estimate these simultaneous needs for larger packet sizes:

application	packet size	overhead
bcp	8192	
Client-Library	8192	
Client-Library	4096	
Client-Library	4096	
Total	24576	
Multiply by 3 buffers / user	*3	
	73728	
Compute 2% overhead and		* .02=1474
round up to a multiple of 512		1536
Add overhead	+ 1536	
Total additional network memory	75264	

You should set **additional netmem** to 75264.

See **bcp** and **isql** in the *SQL Server Utility Programs* manual for your platform for information on using larger packet sizes from these programs. Open Client Client-Library documentation includes information on using variable packet sizes.

Choosing Packet Sizes

The **default network packet size** and **maximum network packet size** variables must be multiples of 512. In addition, you should choose a Server packet size that works efficiently with the underlying packet size on your network for best performance. The two criteria are:

- Reducing the number of server reads and writes to the network
- Reducing unused space in network packets (increasing network throughput)

For example, if your network packet size carries 1500 bytes of data, setting SQL Server's packet size to 1024 (512*2) will probably achieve better performance than setting it to 1536 (512*3).

Figure 12-2: Factors in Determining Packet Size shows sample packet sizes.

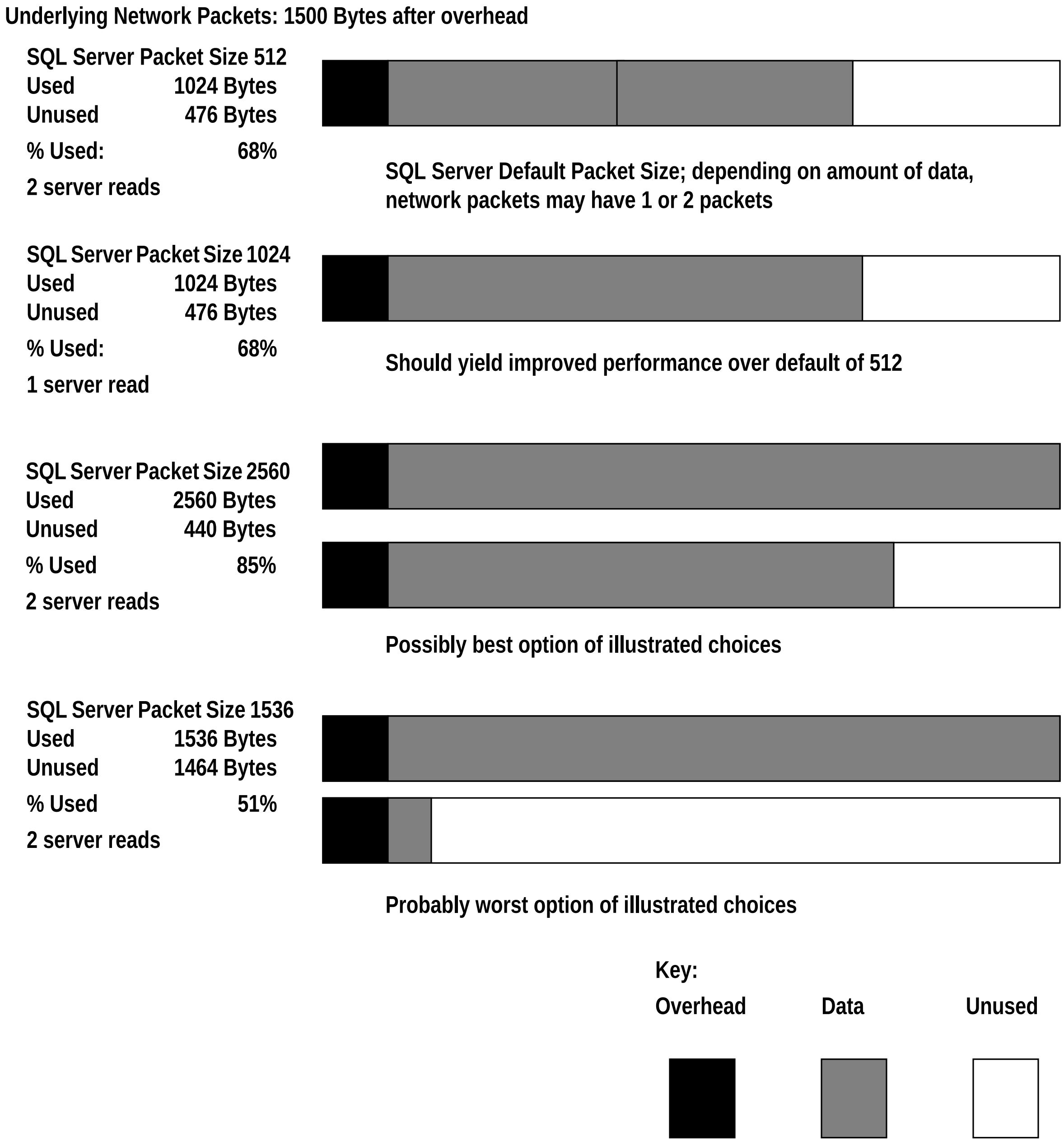


Figure 12-2: Factors in Determining Packet Size

After you determine the available data space of the underlying packets on your network, you should perform your own benchmark tests to determine the optimum size for your configuration.

extent i/o buffers

extent i/o buffers allocates the specified number of extents (8 data pages) for use by the **create index** command. The default value is 0, which means that SQL Server reads and writes individual pages for intermediate sort results and index pages to disk, one page at a time. Allocating extent I/O buffers speeds up indexing by speeding up disk reads and writes.

All of the buffers are allocated to the first user who creates an index. If multiple users create indexes simultaneously, the first user to start uses extent I/Os; all others uses normal page I/Os. You should try to schedule large index creation at times when few other users are on the system.

For each extent I/O buffer you configure, the server allocates 8 data pages from memory allocated to the server with the **memory** configuration variable. A reasonable value for a server with 8MB of memory and a 20% procedure cache is 10, requiring 160K of memory (10 buffers * 8 pages per buffer * 2048 bytes per page). On Stratus, the total is 320K, 10 buffers * 8 pages * 4K data pages.

identity burning set factor

IDENTITY columns are columns of type *numeric* and scale zero whose value is generated by SQL Server. Column values can range from a low of 1 to a high determined by the column precision.

For each table with an IDENTITY column, SQL Server divides the set of possible column values into blocks of consecutive numbers, and makes one block at a time available in memory. Each time you insert a row into a table, SQL Server automatically assigns the IDENTITY column the next available value from the block. When all the numbers in a block have been used, the next block becomes available.

This method of choosing IDENTITY column values improves server performance. When SQL Server assigns a new column value, it reads the current maximum value from memory and adds 1. Disk access becomes necessary only after all values within the block have been used. Because all remaining numbers in a block are discarded in the event of server failure, this method can lead to gaps in IDENTITY column values.

Use **identity burning set factor** to change the percentage of potential column values that is made available in each block. This number should be high enough for good performance, but not so high that gaps in column values are unacceptably large. The default value, 5000, releases .05% of the potential IDENTITY column values for use at a time.

To get the correct value for **sp_configure**, express the percentage in decimal form, then multiply it by 10^7 (10,000,000). For example, to release 15 percent (.15) of the potential IDENTITY column values at a time, you specify a value of .15 times 10^7 (or 1,500,000) in **sp_configure**:

```
sp_configure "identity burning set factor", 1500000
```

Improving Performance Using Segments

In a large, multi-database and / or multi-drive SQL Server environment, careful attention to the allocation of space to databases and to the placement of database objects on physical devices can enhance system performance. Preferably, each database will have exclusive use of database devices, that is, it won't share a physical disk with another database.

Generally, placing a table on one physical device, its nonclustered index on a second physical device, and the transaction log on a third physical device can speed performance. The **log on** extension to **create database** (or **sp_logdevice**) handles the placement of the transaction log on a separate physical disk. Placing other database objects on specific physical devices is managed through the use of segments, which provide a technique for labeling collections of database devices. (Placing tables and indexes on segments is explained in Chapter 3, "Managing Physical Resources".)

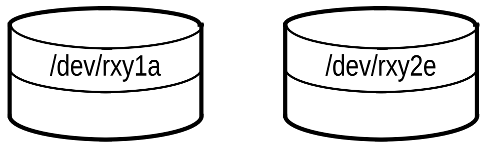
Placing objects on segments may improve performance since default placement of objects can split tables and indexes across devices. In particular, a well-tuned database restricts indexes to a single device.

Splitting a Large Table Across Segments

Performance can be improved for high-volume multi-user applications when large tables are split across segments that are located on separate disk controllers.

Figure 12-3: Splitting a Large Table Across Two Segments illustrates the process of splitting a table across two segments. First, the devices are initialized with **disk init**, and then assigned to the database with **alter database**. Two segments are created, one on each device. A third segment is created and then extended, so that it labels both devices. The *system* and *default* segments are dropped from both of the devices.

Physical devices



Select physical devices to be used by SQL Server.

use ***master***

Start in **master** *database.*

disk init	disk init
name = "mydisk1",	**name** = "mydisk2",
physname = "/dev/rxy1a",	**physname** = "/dev/rxy2e",
vdevno = 7,	**vdevno** = 8,
size = 2048	**size** = 2048

Map SQL Server database device name to physical device with **disk init***.*

alter database mydata
on mydisk1 = 4, mydisk2 = 4

Add the devices **mydisk1** *and* **mydisk2** *to* **mydata***.*

use mydata

Change to **mydata** *database.*

sp_addsegment seg_mydisk1, mydata, mydisk1
sp_addsegment seg_mydisk2, mydata, mydisk2
sp_addsegment seg_bothdisks, mydata, mydisk1
sp_extendsegment seg_bothdisks, mydata, mydisk2

Add a segment on **mydisk1** *and another on* **mydisk2***. Create a third segment, and extend it to span both disks.*

sp_dropsegment "default", mydata, mydisk1
sp_dropsegment system, mydata, mydisk1
sp_dropsegment "default", mydata, mydisk2
sp_dropsegment system, mydata, mydisk2

Drop **system** *and* **default** *segments from the new devices.*

create table authors (au_id etc.) **on** seg_mydisk1
create clustered index au_ind **on** authors (au_id)
on seg_mydisk1

Create the table and clustered index on the segment.

[use **bcp** to load half of the rows]

Load half of the rows.

sp_placeobject segmydisk2, authors

Place the object on the second segment.

[use **bcp** to load the rest of the rows]

Load the rest of the rows.

sp_placeobject seg_bothdisks, authors

Place the table on the segment that spans both disks.

Figure 12-3: Splitting a Large Table Across Two Segments

The table and its clustered index are created on the first segment, and half of the data is loaded. Then, **sp_placeobject** is used to cause all further allocations of disk space to occur on the second segment. Once the data is loaded on the second segment, **sp_placeobject** is used again, this time placing the table on the segments that span both devices.

➤ ***Note***

The order of events is quite important at certain stages. In particular, the clustered index **must** be created before the table is placed on the second segment. After that point, creation of a clustered index would cause the entire object to migrate to the second segment.

If the index is created after the creation of *seg_bothdisks*, the allocation of disk space is unpredictable.

If the table is updated frequently, the balance of disk allocation may change over time. To guarantee that the speed advantages are maintained, it may be necessary to drop and recreate the table.

13 Locking

Introduction

SQL Server protects the tables or data pages currently used by active transactions by locking them. Locking is a concurrency control mechanism: it ensures the consistency of data across transactions. It is needed in a multi-user environment, since several users may be working with the same data at the same time.

This chapter discusses:

- The types of consistency issues that arise in multi-user databases, and the SQL Server options for enforcing different levels of consistency
- Locks used in SQL Server
- Using the **set transaction isolation level** command to meet ANSI requirements
- Locks used by Transact-SQL commands
- System procedures for examining locks and user processes blocked by locks (**sp_lock** and **sp_who**)
- SQL Server's handling of deadlocks
- Locking and performance issues
- Strategies for reducing lock contention

Overview of Locking

Consistency of data means that if multiple users repeatedly execute a series of transactions, the results are the same each time. This means that simultaneous retrievals and modifications of data do not interfere with each other.

For example, assume that the following transactions, T1 and T2, run at approximately the same time:

T1	Event Sequence	T2
`begin transaction`	T1 and T2 start	`begin transaction`
`update account` `set balance = balance - 100` `where acct_number = 25`	T1 updates balance for one account by subtracting $100	
	T2 queries the sum balance for accounts which is off by $100	`select sum(balance)` `from account` `where acct_number < 50`
	T2 ends	`commit transaction`
`update account` `set balance = balance + 100` `where acct_number = 45`	T1 updates balance of other account to add the $100	
`commit transaction`	T1 ends	

Figure 13-1: Consistency Levels in Transactions

If transaction T2 runs before T1, or after T1, both executions would return the same value. But if T2 could run in the middle of transaction T1 (after the first **update**), the result for transaction T2 would be different by 100. SQL Server prevents this possibility by locking the data used in T1 until the transaction finishes, and only then allowing T2 to complete its query.

Locking is handled automatically by SQL Server and is not under user control (with a few exceptions described later). However, you must still know how and when to use transactions to preserve the consistency of your data, while maintaining high performance and throughput. Transactions are described in the *Transact-SQL User's Guide* and in Volume 1 of the *SQL Server Reference Manual*.

Isolation Levels and Transactions

The ANSI standard defines 3 levels of isolation for SQL transactions. Each isolation level specifies the kinds of actions that are not permitted while concurrent transactions execute. Higher levels include the restrictions imposed by the lower levels:

- Level 1 prevents **dirty reads**. “Dirty reads” occur when one transaction modifies a row, and then a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid.

  The example in Figure 13-1 shows a case of a “dirty read”. The following example describes another case:

T3	Event Sequence	T4
`begin transaction`	T3 and T4 start	`begin transaction`
`update account` `set balance = balance - 100` `where acct_number = 25`	T3 updates balance for one account by subtracting $100	
	T4 queries the current sum balance for accounts	`select sum(balance)` `from account` `where acct_number < 50`
	T4 ends	`commit transaction`
`rollback transaction`	T3 rolls back invalidating the results from T4	

Figure 13-2: Dirty Reads in Transactions

If transaction T4 queries the table after T3 updates it, but before it rolls back the change, the amount calculated by T4 is off by 100.

- Level 2 prevents **non-repeatable reads**. "Non-repeatable reads" occur when one transaction reads a row and then a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read. For example:

T5	Event Sequence	T6
`begin transaction`	T5 and T6 start	`begin transaction`
`select balance` `from account` `where acct_number = 25`	T5 queries the balance for one account	
	T6 updates the balance for that same account	`update account` `set balance = balance - 100` `where acct_number = 25`
	T6 ends	`commit transaction`
`select balance` `from account` `where acct_number = 25`	T5 makes same query as before and gets different results	
`commit transaction`	T5 ends	

Figure 13-3: Non-repeatable Reads in Transactions

If transaction T6 modifies and commits the changes to the *account* table after the first query in T5 but before the second one, the same two queries in T5 produce different results.

- Level 3 prevents **phantoms**. "Phantoms" occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, or **update**). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows. For example:

T7	Event Sequence	T8
`begin transaction`	T7 and T8 start	`begin transaction`
`select * from account where acct_number < 25`	T7 queries a certain set of rows	
	T8 inserts a row that meets the criteria for query in T7	`insert into account (acct_number, balance) values (19, 500)`
	T8 ends	`commit transaction`
`select * from account where acct_number < 25`	T7 makes the same query and gets a new row	
`commit transaction`	T7 ends	

Figure 13-4: Phantoms in Transactions

If transaction T8 inserts rows into the table that satisfy T7's search condition after the T7 executes the first **select**, subsequent reads by T7 using the same query result in a different set of rows.

By default, SQL Server's transaction isolation level is 1. You can enforce the other isolation levels using the **holdlock** keyword of the **select** statement or using the **transaction isolation level** option of the **set** command. **holdlock** and **transaction isolation level** are described in "holdlock and Isolation Levels" on page 13-9.

Granularity of Locks

The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server could lock as much as the entire database or as little as a row of data. However, such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process each lock) in the server.

By increasing the lock size, the amount of work required to obtain a lock becomes smaller, but large locks can degrade performance as more users must wait until the locks are released. By decreasing the lock size, more of the data becomes accessible to other users, but small locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks. Smaller locks also increase the likelihood of deadlocks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

SQL Server achieves this balance by locking only data pages or tables. It does not lock work or temporary tables because they are always single-user by definition. These locking mechanisms are described in the following sections.

Locking in SQL Server

SQL Server handles all locking decisions. It chooses which type of lock to use after it determines the query plan. However, the way you write a query or transaction can affect the type of lock the server chooses. You can also force the server to choose more or less restrictive locks by including the **holdlock**, **noholdlock**, or **shared** keywords with your queries. Each of these options are described later in this section.

SQL Server has two levels of locking: page locks and table locks. Page locks are generally less restrictive (or smaller) than table locks; one table can span multiple data pages. SQL Server attempts to use page locks as frequently as possible to reduce the contention for data among users and to improve concurrency.

SQL Server uses table locks to provide more efficient locking when it suspects the whole table may be needed by a statement. For example, the following statement generates one table lock instead of numerous page locks:

```
update account set balance = balance * 1.05
```

Table locks also provide a way to avoid lock collisions at the page level. SQL Server automatically uses table locks for some commands.

SQL Server first tries to satisfy requests with page locks whenever possible. However, once a statement accumulates more than 200 page locks, SQL Server attempts to issue a table lock on that object. If it succeeds, the page locks are no longer necessary and are released.

Page Locks

The following describes the different types of page locks:

- **Shared**

  SQL Server applies shared locks for read operations. If a shared lock has been applied to a data page, other transactions can also acquire a shared lock even when the first transaction is not finished. However, no transaction can acquire an exclusive lock on the page until all shared locks on it are released. That is, many transactions can simultaneously read the page, but no one can write to it.

  By default, SQL Server releases shared page locks after the statement completes; it does not hold them until the end of its transaction. Transactions that need an exclusive page lock wait or "block" for the release of the shared page locks before continuing.

- **Exclusive**

  SQL Server applies exclusive locks for data modification operations. When a transaction gets an exclusive lock, other transactions cannot acquire a lock of any kind on the page until the exclusive lock is released at the end of its transaction. Those other transactions wait or "block" until the exclusive lock is released, before continuing.

- **Update**

  SQL Server applies update locks during the initial portion of an **update** or **delete** operation when the pages are being read. The update locks allow shared locks on the page, but do not allow other update locks or exclusive locks. This is an internal lock to help avoid deadlocks. Later, if the pages need to be changed and no other shared locks exist on the pages, the update locks are promoted to exclusive locks.

In general, read operations acquire shared locks, while write operations acquire exclusive locks. However, SQL Server can only apply page-level exclusive and update locks if the column used in the search argument is part of an index.

The following examples show what kind of page locks SQL Server uses for the respective statement (assuming indexes are used in their search arguments):

`select balance from account where acct_number = 25`	*Shared Page Lock*
`insert account values(34, 500)`	*Exclusive Page Lock*
`delete account where balance < 0`	*Update Page Locks* *Exclusive Page Locks*
`update account set balance = 0 where acct_number = 25`	*Update Page Lock,* *Exclusive Page Lock*

Table Locks

The following describes the different types of table locks.

- **Intent**

  An intent lock indicates that certain types of page-level locks are currently held in a table. SQL Server applies an intent table lock with each shared or exclusive page lock. Setting an intent lock prevents other transactions from subsequently acquiring a shared or exclusive lock on the table which contains that locked page. Intent locks are held as long as the concurrent page locks are in effect.

- **Shared**

  This lock is similar to the shared page lock, except that it affects the entire table. For example, SQL Server applies shared table locks is with the **create nonclustered index** statement.

- **Exclusive**

  This lock is similar to the exclusive page lock, except that it also affects the entire table. SQL Server applies exclusive table locks during the **create clustered index** command, for example. **update** or **delete** statements generate exclusive table locks if their search arguments do not reference indexed columns of the object.

The following examples show the respective page and table locks issued for each statement:

`select balance from account where acct_number = 25`	*Intent shared table lock,* *shared page lock*
`insert account values(34, 500)`	*Intent exclusive table lock,* *exclusive page lock*

These two examples assume there are indexes on the *balance* and *acct_number*; otherwise, SQL Server would just issue exclusive table locks for the statements:

`delete account` `where balance < 0`	*Intent exclusive table lock, update page locks, exclusive page locks*
`update account set balance = 0` `where acct_number = 25`	*Intent exclusive table lock, update page lock, exclusive page lock*

Demand Locks

Demand locks prevent any more shared locks from being set. SQL Server sets a demand lock to indicate that a transaction is next in line to lock a table or page. This avoids situations in which read transactions acquire overlapping shared locks, monopolizing a table or page, so that a write transaction waits indefinitely for its exclusive lock.

After waiting on several different read transactions, SQL Server gives a demand lock to the write transaction. As soon as the existing read transactions finish, the write transaction is allowed to proceed. Any new read transactions must then wait for the write transaction to finish, when its exclusive lock is released.

holdlock and Isolation Levels

The **holdlock** keyword, used in **select** and **readtext** statements, makes a shared page or table lock more restrictive. It applies:

- To shared locks
- To the table or view for which it is specified
- For the duration of the statement or transaction containing the statement

In a transaction, **holdlock** instructs SQL Server to hold the shared lock that it has set until the completion of that transaction, instead of releasing the lock as soon as the required table, view, or data page is no longer needed (whether or not the transaction has completed). SQL Server always holds exclusive locks until the end of a transaction.

When you use **holdlock** with a statement, SQL Server applies shared page locks if the statement's search argument references indexed columns of the object. Otherwise, SQL Server applies a shared table lock. The following example assumes that an index does not exist for *acct_number*:

Shared Table Lock

```
select balance
from account holdlock
where acct_number = 25
```

SQL Server's default handling of shared locks—releasing page locks as soon as the table or view is no longer needed—allows concurrent access to the database even while a lengthy transaction is proceeding. However, it only enforces ANSI isolation level 1, which prevents "dirty reads". By using the **holdlock** keyword, you can enforce isolation level 3, which prevents "non-repeatable reads" and "phantoms".

Preventing Dirty Reads

At isolation level 1 (the default), SQL Server prevents "dirty reads" by:

- Applying exclusive locks on pages or tables being changed. It holds those locks until the end of the transaction.
- Applying shared locks on pages being searched. It releases those locks after processing the object.

For example, let's look at the previous case of a "dirty read":

T3	Event Sequence	T4
`begin transaction`	T3 and T4 start	`begin transaction`
`update account` `set balance = balance - 100` `where acct_number = 25`	T3 updates account after getting exclusive lock	
	T4 tries to get shared lock to query account but must wait until T3 releases its lock	`select sum(balance)` `from account` `where acct_number < 50`
`rollback transaction`	T3 ends and releases its exclusive lock	
	T4 gets shared lock, queries account, and ends	`commit transaction`

Figure 13-5: Avoiding Dirty Reads in Transactions

When the **update** statement in transaction T3 executes, SQL Server applies an exclusive lock (a page-level lock if *acct_number* is indexed or a table-level lock otherwise) on *account*. The query in T4 cannot execute until the lock is released, when T3 ends with the **rollback**.

Preventing Non-Repeatable Reads and Phantoms

At isolation level 3, SQL Server also prevents "non-repeatable reads" and "phantoms" by (the restrictions imposed by level 2 are included in level 3):

- Applying exclusive locks on pages or tables being changed
- Applying shared locks on pages or tables being searched

It holds both exclusive and shared locks until the end of the transaction.

For example, let's look at the previous case of a "phantom":

T7	Event Sequence	T8
`begin transaction`	T7 and T8 start	`begin transaction`
`select * from account holdlock where acct_number < 25`	T7 queries account and holds acquired shared locks	
	T8 tries to insert row but must wait until T7 releases its locks	`insert into account (acct_number, balance) values (19, 500)`
`select * from account holdlock where acct_number < 25`	T7 makes same query and gets same results	
`commit transaction`	T7 ends and releases its shared locks	
	T8 gets its exclusive lock, inserts new row, and ends	`commit transaction`

Figure 13-6: Avoiding Phantoms in Transactions

In transaction T7, SQL Server applies shared page locks (if an index exists on the *acct_number* argument) or a shared table lock (if no index exists) and holds those locks until the end of T7. The **insert** in T8 cannot get its exclusive lock until T7 releases those shared locks.

Defining the Default Isolation Level

The ANSI standard requires a default transaction isolation level of 3. To enforce this default, Transact-SQL provides the option **transaction isolation level** with the **set** statement. For example, you can make level 3 the default isolation level for a session as follows:

```
set transaction isolation level 3
```

This instructs SQL Server to automatically apply a **holdlock** to all **select** operations in a transaction. For more information about the **transaction isolation level** option, see "Transactions" in Volume 1 of the *SQL Server Reference Manual*.

Using the **noholdlock** *Keyword*

The **noholdlock** keyword prevents SQL Server from holding any shared locks acquired during the execution of the **select** statement, regardless of the transaction isolation level currently in effect. **noholdlock** is useful in situations when the default isolation level is 3, but your query does not need to hold its shared locks until the end of the transaction.

Cursors and Locking

Cursor locking methods are similar to the other locking methods for SQL Server. For cursors declared as **read only** or declared without the **for update** clause, SQL Server uses shared page locks on the data page that includes the current cursor position. For cursors declared with **for update**, SQL Server uses update page locks by default when scanning tables or views referenced with the **for update** clause of **declare cursor.** If the **for update** list is empty, all tables and views referenced in the **from** clause of the *select_statement* receive update locks.

SQL Server releases shared and update locks when the cursor position moves off a data page. If a row of an updatable cursor is updated or deleted, SQL Server promotes its shared or update lock to an exclusive lock. Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared or update locks when using the **holdlock** keyword or the **set isolation level 3** option.

The following describes the locking behavior for cursors at each isolation level:

- At level 1, SQL Server uses a shared or update lock on base table pages that contain a row representing a current cursor position.

The page remains locked until the current cursor position moves off the page as a result of **fetch** statements.

- At level 3, SQL Server uses a shared or update lock on any base table pages which have been read in a transaction through the cursor. SQL Server holds the locks until the transaction ends; it does not release the locks when the data page is no longer needed.

If you do not set the **close on endtran** option, a cursor remains open past the end of the transaction, and its current page lock remains in effect. It could also continue to acquire locks as it fetches additional rows.

Using the **shared** *Keyword*

When declaring an updatable cursor using the **for update** clause, you can tell SQL Server to use shared page locks (instead of update page locks) in the cursor's **declare cursor** statement:

```
declare cursor_name cursor
   for select select_list
   from {table_name | view_name} shared
   for update [of column_name_list]
```

This allows other users to obtain an update lock on that table or view. You can only use **shared** with the **declare cursor** statement.

You can use the **holdlock** keyword in conjunction with **shared** after each table or view name, but **holdlock** must precede **shared** in the **select** statement. For example:

```
declare authors_crsr cursor
for select au_id, au_lname, au_fname
    from authors holdlock shared
    where state != 'CA'
    for update of au_lname, au_fname
```

These are the effects of specifying the **holdlock** or **shared** options (of the **select** statement) when defining an updatable cursor:

- If you omit both options, you can read data on the currently fetched pages only. Other users cannot update, through a cursor or otherwise, your currently fetched pages. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages.
- If you specify the **shared** option, you can read data on the currently fetched pages only. Other users cannot update, through a cursor or otherwise, your currently fetched pages.

- If you specify the **holdlock** option, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update, through a cursor or otherwise, your currently fetched pages or pages fetched in your current transaction. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages or pages fetched in your current transaction.
- If you specify both options, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update, through a cursor or otherwise, your currently fetched pages.

Summary of Lock Types and Lock Limits

The following table describes the types of locks SQL Server applies for **insert** and **create index** statements:

Statement	Table Lock	Page Lock
insert	IX	X
create clustered index	X	-
create nonclustered index	S	-
IX = intent exclusive, S = shared, X = exclusive		

Table 13-1: Summary of Locks for insert and create index Statements

This next table describes the types of locks SQL Server applies for **select**, **delete**, and **update** statements. It divides the **select**, **update**, and **delete** statements into two groups, since the types of locks they use can vary if the statement's search argument references indexed columns on the object:

Statement	Indexed		Not Indexed	
	Table Lock	Page Lock	Table Lock	Page Lock
select	IS	S	IS	S
select with holdlock	IS	S	S	-
update	IX	U, X	X	-
delete	IX	U, X	X	-
IS = intent shared, IX = intent exclusive, S = shared, U = update, X = exclusive				

Table 13-2: Summary of Locks for select, update and delete Statements

Note that the above tables do not describe situations in which SQL Server initially uses table locks (if a query requires the entire table), nor when it promotes to a table lock after 200 page locks.

Configuring SQL Server's Lock Limit

Each lock counts toward SQL Server's limit of total number of locks. By default, SQL Server is configured with 5000 locks. A System Administrator can change this limit using the **sp_configure** system procedure. For example:

```
sp_configure locks, 10000
```

You may also need to adjust the **memory** option of **sp_configure**, since more locks use additional memory.

The number of locks required by a server can vary depending on the number of concurrent processes and the types of actions performed by the transactions. However, a general assumption is that each concurrent process uses about 20 locks. For more information about **sp_configure**, see Chapter 12, ''Fine-Tuning Performance and Operations''.

Example of Locking

This section describes the sequence of locks applied by SQL Server for the two transactions used at the beginning of this chapter:

T1	Event Sequence	T2
`begin transaction`	T1 and T2 start	`begin transaction`
`update account` `set balance = balance - 100` `where acct_number = 25`	T1 gets exclusive lock and updates account	
	T2 tries to query account but must wait until T1 ends	`select sum(balance)` `from account` `where acct_number < 50`
`update account` `set balance = balance + 100` `where acct_number = 45`	T1 keeps updating account and gets more exclusive locks	
`commit transaction`	T1 ends and releases its exclusive locks	
	T2 gets shared locks, queries account, and ends	`commit transaction`

Figure 13-7: Locking Example Between Two Transactions

The following sequence of locks assumes an index exists on the *acct_number* column of the *account* table, a default isolation level of 1, and 10 rows per data page (50 rows divided by 10 equals 5 data pages):

T1 Locks	T2 Locks
Update lock page 1	
Exclusive lock page 1	Shared lock page 1 denied, wait for release
Intent exclusive table lock on account	
Update lock page 5	
Exclusive lock page 5	
Release all locks at commit	
	Shared lock page 1, release lock page 1
	Intent shared table lock on account
	Shared lock page 2, release lock page 2
	Shared lock page 3, release lock page 3
	Shared lock page 4, release lock page 4
	Shared lock page 5, release lock page 5
	Release intent shared table lock

Table 13-3: Sequence of Locks at Isolation Level 1

If no index exists for *acct_number*, SQL Server applies exclusive table locks for T1 instead of page locks:

T1 Locks	T2 Locks
Exclusive table lock on account	Shared lock page 1 denied, wait for release
Release exclusive table lock at commit	
	Shared lock page 1, release lock page 1
	Intent shared table lock on account
	Shared lock page 2, release lock page 2
	Shared lock page 3, release lock page 3
	Shared lock page 4, release lock page 4
	Shared lock page 5, release lock page 5
	Release intent shared table lock

Table 13-4: Sequence of Locks at Isolation Level 1 with No Index

If you add a **holdlock** or make isolation level 3 the default using the **transaction isolation level** option for transaction T2, the lock sequence is as follows (assuming an index exists for *acct_number*):

T1 Locks	T2 Locks
Update lock page 1	
Exclusive lock page 1	Shared lock page 1 denied, wait for release
Intent exclusive table lock on account	
Update lock page 5	
Exclusive lock page 5	
Release all locks at commit	
	Shared lock page 1
	Intent shared table lock on account
	Shared lock page 2
	Shared lock page 3
	Shared lock page 4
	Shared lock page 5
	Release all locks at commit

Table 13-5: Sequence of Locks at Isolation Level 3

If you add **holdlock** or make **transaction isolation level 3** for T2 and no index exists for *acct_number*, SQL Server applies table locks for both transactions instead of page locks:

T1 Locks	T2 Locks
Exclusive table lock on account	Shared table lock denied, wait for release
Release exclusive table lock at commit	
	Shared table lock on account
	Release shared table lock at commit

Table 13-6: Sequence of Locks at Isolation Level 3 with No Index

Viewing Locks with *sp_lock*

To get a report on the locks currently being held on SQL Server, use the system procedure **sp_lock**. For example:

```
sp_lock
```

```
The class column will dispaly the cursor name for locks
associated with a cursor for the current user and the cursor id
for other users.
spid  locktype      table_id    page   dbname  class
----  ------------  ----------  ----   ------  ---------------
1     Ex_intent     1308531695  0      master  Non cursor lock
1     Ex_page       1308531695  761    master  Non cursor lock
5     Ex_intent     144003544   0      userdb  Non cursor lock
5     Ex_page       144003544   509    userdb  Non cursor lock
5     Ex_page       144003544   1419   userdb  Non cursor lock
5     Ex_page       144003544   1420   userdb  Non cursor lock
5     Ex_page       144003544   1440   userdb  Non cursor lock
5     Sh_page       144003544   1440   userdb  Non cursor lock
5     Sh_table      144003544   0      userdb  Non cursor lock
5     Update_page   144003544   1440   userdb  Non cursor lock
4     Ex_table      240003886   0      pubs2   Non cursor lock
4     Sh_intent     112003436   0      pubs2   Non cursor lock
4     Ex_intent-blk 112003436   0      pubs2   Non cursor lock
```

The *locktype* column indicates not only whether the lock is a shared lock ("Sh" prefix), an exclusive lock ("Ex" prefix) or an "update" lock, but also whether it is held on a table ("table" or "intent"), or on a "page".

A "blk" suffix indicates that this process is blocking another process which needs to acquire a lock. As soon as this process completes, the other process(es) move forward. A "demand" suffix indicates that a process will acquire an exclusive lock as soon as all current shared locks are released.

Information about Blocked Processes in *sp_who*

The system procedure **sp_who** reports on system processes. If a user's command is being blocked by locks held by another process:

- The *status* column shows "lock sleep"
- The *blk* column shows the process ID of the process that holds the lock or locks

Deadlocks and Concurrency in SQL Server

A deadlock occurs when two users each have a lock on a separate object. Each wants to acquire an additional lock on the other user's object. When this happens, the first user is waiting for the second to let go of the lock, but the second user will not let it go until the lock on the first user's object is released.

For example:

T9	Event Sequence	T10
`begin transaction`	T9 and T10 start	`begin transaction`
`update savings` `set balance = balance - 250` `where acct_number = 25`	T9 gets exclusive lock for savings while T10 gets exclusive lock for checking	`update checking` `set balance = balance - 75` `where acct_number = 45`
`update checking` `set balance = balance + 250` `where acct_number = 45`	T9 waits for T10 to release its lock while T10 waits for T9 to release its lock; deadlock occurs	`update savings` `set balance = balance + 75` `where acct_number = 25`
`commit transaction`		`commit transaction`

Figure 13-8: Deadlocks in Transactions

If transactions T9 and T10 execute simultaneously and both transactions acquire exclusive locks with their initial **update** statements, they become deadlocked waiting for each other to release their locks, which won't happen until the **commit transaction**.

SQL Server detects deadlocks and chooses the user whose process has accumulated the least amount of CPU time as the victim. SQL Server rolls back that user's transaction, notifies the application program of this action with message number 1205, and allows the other users' processes to move forward.

In a multi-user situation, each user's application program should check every transaction for message 1205. It indicates that the user was selected as the victim of a deadlock. The application program must restart that transaction.

Avoiding Deadlocks

It is possible to encounter deadlocks when many long-running transactions are executed at the same time in the same database. Deadlocks become more common as the lock contention increases between those transactions (decreasing concurrency). Reducing lock contention, such as avoiding table locks and not holding shared locks, is described at the end of the chapter.

Well-designed applications can avoid deadlocks by always acquiring locks in the same order. Updates to multiple tables should always be performed in the same order, with no table updated twice.

For example, the transactions described in Table 13-8 could avoid their deadlock by updating either the *savings* or *checking* table first in both transactions. That way, one transaction gets the exclusive lock first and proceeds, while the other transaction waits to receive its exclusive lock on the same table when the first transaction ends.

SQL Server also avoids deadlocks by applying the following locks:

- Update page locks permit only one exclusive page lock at a time. Other update or exclusive locks must wait until that exclusive lock is released before accessing the page. However, update locks affect concurrency since the net effect is that all updates on a page happen only one at a time.
- Intent table locks act as a place holder when shared or exclusive page locks are acquired. They inform other transactions that need table locks whether or not they can be granted without having the server scan the page lock queue. It also helps avoid lock collisions between page level locks and table level locks.

Locking and Performance of SQL Server

How locking affects the performance of SQL Server is directly related to the issue of concurrency. An increase in the number of concurrent users of a server may also increase lock contentions which decreases performance. Locks affect performance when:

- Processes wait for locks to be released

  Any time a process waits for another process to complete its transaction and release its locks, the overall response time and throughput is affected. This is what we mean by lock contention.

- Transactions result in frequent deadlocks

  As described earlier, any deadlock causes one transaction to be aborted which must then be restarted by the application; this severely affects the throughput of applications. Deadlocks cannot be completely avoided. However, redesigning the way transactions access the data can help reduce their frequency.

- Creating indexes locks tables

  Creating a clustered index locks all users out of the table until the index is created. Creating a nonclustered index locks out all updates until it is created. Either way, you should create indexes at time when there is little other activity on your server.

The next section provides some tips on reducing lock contention to increase your server's performance.

Reducing Lock Contention

Redesigning the tables that have the highest lock contention may also improve performance. For example, assume a table has no index on one of its columns. Any update specifying that column in its search argument generates a table lock. Table locks generate more lock contentions than page locks, since no other process can then access the table. By creating an index on that column, those updates first use page locks, improving the concurrency in accessing the table.

Another way to reduce contention is by decreasing the **fillfactor** of your indexes. **fillfactor** (defined by either **sp_configure** or **create index**) determines how full SQL Server makes each data page when it is creating a new index on existing data. When there is more empty space in the index and leaf pages, the chances of lock contention are reduced.

As the keys are spread out over more pages, it becomes more likely that a page you want is not the page someone else also needs. In addition, since **fillfactor** helps reduce page splits, exclusive locks are also minimized on the index, improving performance. For more information about **fillfactor**, see Chapter 12, ''Fine-Tuning Performance and Operations''.

The following provides additional guidelines to reducing lock contention:

- Avoid "hot spots."

  "Hot spots" occur when all updates take place at a certain page, as in a "heap" (non-indexed table) where all additions happen at the last page. For example, a history table with no index which is updated by everyone will always have lock contentions on the last page.

  One solution is to create separate history tables for various groups of users, which can reduce the wait on the page. You can also create a clustered index to distribute the data across the table.

- Avoid writing transactions which include user interaction.

Since SQL Server holds some locks until the transaction commits, if a user can hold up the commit (even for a short time), there will be higher lock contention.

- Keep transactions short.

  The longer the transaction, the longer exclusive or update locks are held. This blocks other activity and leads to more possible deadlocks.

- Keep transactions in one batch.

  Network interaction during a transaction can introduce unnecessary delays in completing the transaction and releasing its locks.

- Use **holdlock** only when necessary.

  Updates by other transactions may be delayed until a transaction with a **holdlock** releases its shared lock at the end of the transaction. Use **holdlock** only to enforce isolation level 3 when "non-repeatable reads" or "phantoms" may interfere with your results.

14 Managing Multiprocessor Servers

Introduction

SQL Server implements a Virtual Server Architecture, enabling it to take advantage of the parallel processing feature of symmetric multiprocessing (SMP) systems. SQL Server can be run as a single process or as multiple cooperating processes, depending on the number of CPUs available and the demands placed on the server. This chapter describes:

- The target machine architecture for the SMP SQL Server
- SQL Server architecture for SMP environments
- SQL Server task management in the SMP environment
- Managing multiple engines
- Application design issues

Definitions

Here are the definitions of several terms used in this chapter:

- **Process**: An execution environment scheduled onto physical CPUs by the operating system.
- **Engine**: A process running a SQL Server that communicates with the other SQL Server processes via shared memory. An engine can be thought of as one CPU's worth of processing power. It does **not** represent a particular CPU. Also referred to as "server engine".
- **Task**: An execution environment within the SQL Server scheduled onto engines by the SQL Server.
- **Affinity**: Describes a process in which a certain SQL Server task runs only on a certain engine, or that a certain engine runs only on a certain CPU.

Target Architecture

The SMP environment product is intended for machines with the following features:

- A symmetric multiprocessing operating system

- Shared memory over a common bus
- 1-32 processors
- No master processor
- Very high throughput

SQL Server consists of one or more cooperating processes (called **engines**), all running the server program in parallel. See Figure 14-1.

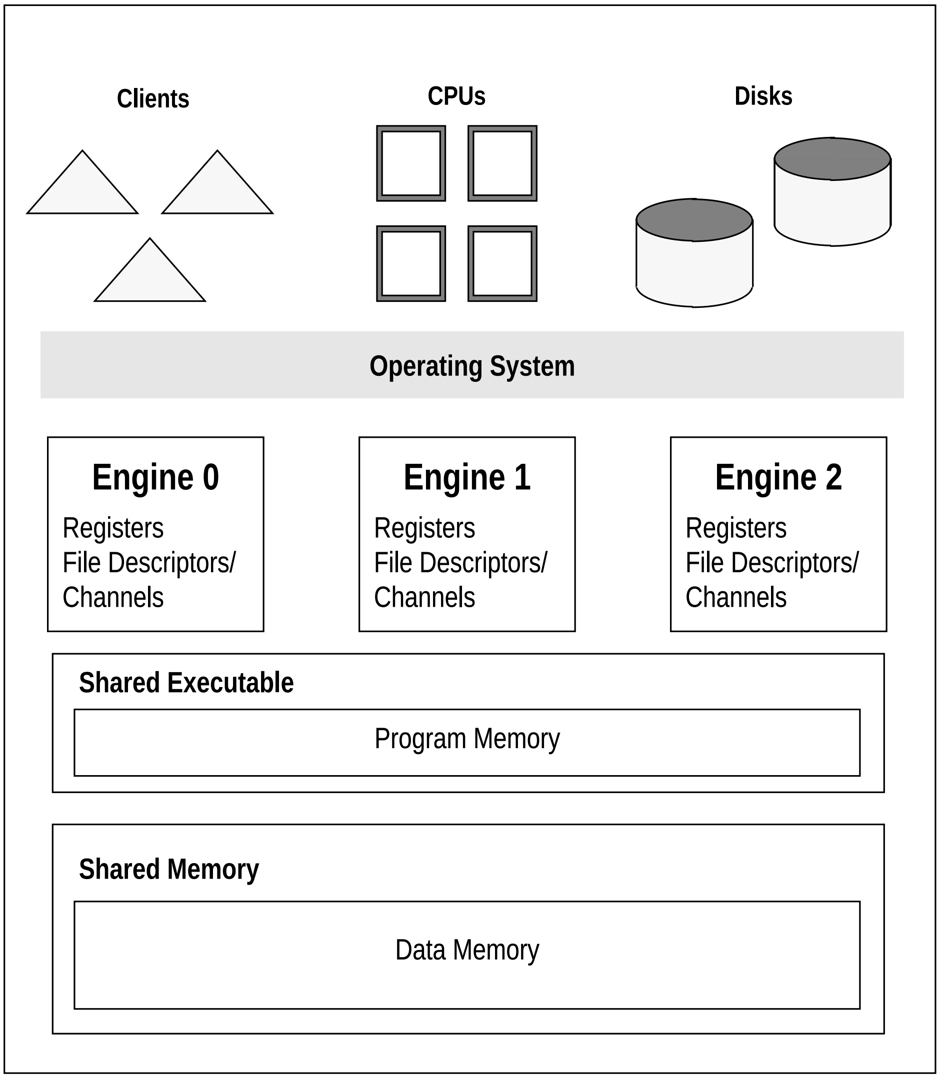


Figure 14-1: SMP Environment Architecture

Only one of the engines, engine 0, handles tasks involving network management; otherwise, all engines are peers. The engines communicate via shared memory.

The server engines perform all database functions, including updates and logging. SQL Server, not the operating system, dynamically schedules client tasks onto available engines. When an engine becomes available, it runs any runnable task.

The operating system schedules the engine processes onto physical processors. Any available CPU is used for any engine; there is no **affinity** between engines and CPUs. The processing is called **symmetric** because the lack of affinity between processes and CPUs creates a symmetrically balanced load

SQL Server Task Management for SMP

Figure 14-2: SQL Server Task Management in the SMP Environment illustrates SQL Server task management. Here is a brief description of the process:

1. A client application issues a login request. In response, SQL Server creates a user task to handle work from the client.
2. The client presents SQL Server with work to do, that is, a series of Transact-SQL commands.
3. SQL Server adds the client's user task to the runnable task queue. The server engines compete for the user task at the head of the task queue.
4. The server engine that takes the user task from the queue converts the Transact-SQL commands into low level steps, such as disk I/O.
5. The engine executes each step until the task completes or blocks while waiting for I/O or locking. When the task blocks, it yields the server engine to run other user tasks. Once the block is resolved (i.e., disk I/O completes or a lock is acquired), the user task is again added to the runnable task queue.
6. After the task blocks for the last time, it continues executing until it finishes. At that time the user task yields the server engine and moves to the sleeping task queue until the client presents the server with more work.

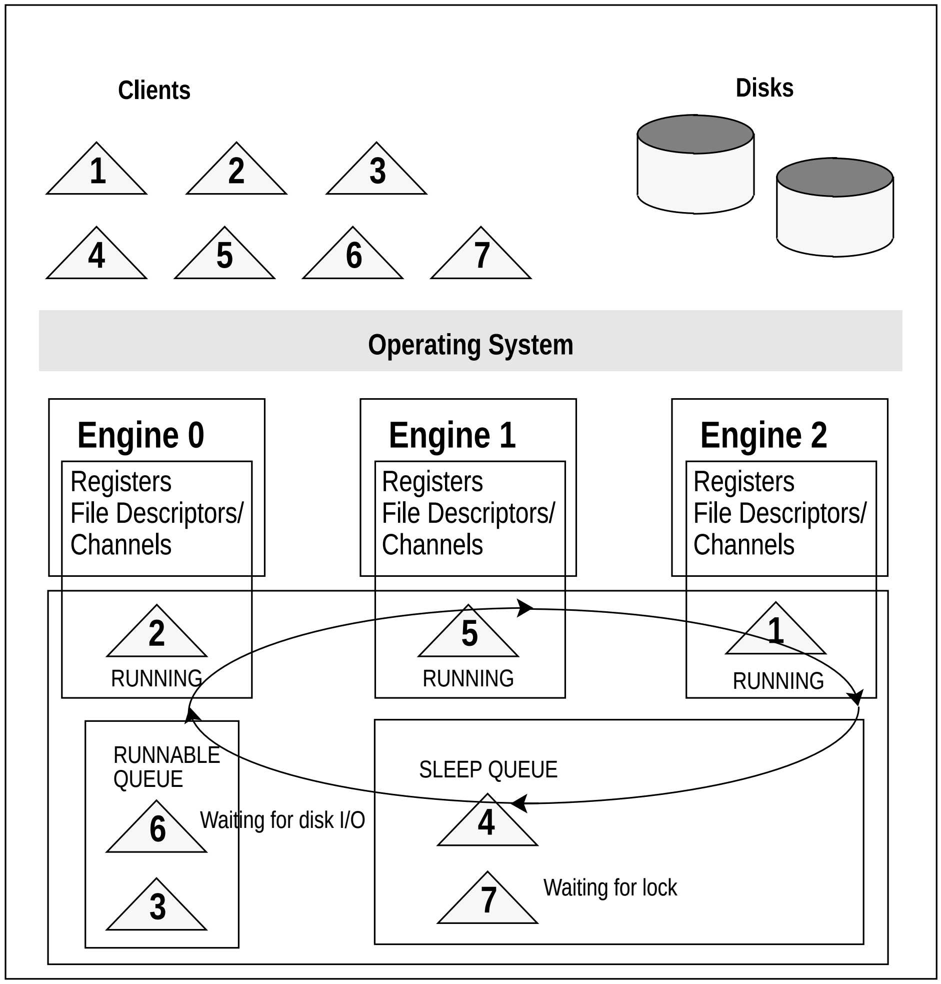


Figure 14-2: SQL Server Task Management in the SMP Environment

The SMP SQL Server is designed in such a way that applications and users see a single database service, no matter how many engines and processors there are.

Configuring an SMP Environment

Configuration of the SMP environment is much the same as in the uniprocessor environment, although SMP machines are typically bigger and handle many more users. The SMP environment provides the additional ability to control the number of engines.

Managing Engines

To achieve optimum performance from an SMP system, you must maintain the right number of engines.

An engine represents a certain amount of CPU power. It is a configurable resource like memory. An engine does not represent a particular CPU.

Resetting the Number of Engines

When you create a new master device (specifically, when you run **buildmaster**), or when you start an SMP server for the first time on a database that has been upgraded from an earlier release, the system is configured for a single engine. To engage multiple engines, you must reset the number of engines the first time you start the server. You may also want to reset the number of engines at other times.

For example:

- You might want to **increase** the number of engines if current performance is not adequate for an application **and** there are enough CPUs on the machine.
- You might want to **decrease** the number of engines if a hardware failure disables CPUs on the machine.

The **max online engines** configuration variable controls the number of engines. Reset this configuration variable with the **sp_configure** system procedure. For example, to set the number of engines to 3:

1. Issue the following command:

   ```
   sp_configure "max online engines", 3
   ```

2. Run the Transact-SQL **reconfigure** command
3. Stop and restart the server.

Repeat these steps whenever you need to change the number of engines. Engines other than engine 0 are brought on line after recovery is complete.

The **min online engines** configuration variable is not used in this release, and you cannot currently specify the minimum number of engines.

Choosing the Right Number of Engines

It is important to choose the right number of engines. Here are some pointers for choosing how many engines to use:

- **Never** have more engines than CPUs. Doing so may slow performance. If a CPU goes off line, use **sp_configure** to reduce the **max online engines** configuration variable by one, run **reconfigure**, and restart the server.
- Have only as many engines as **usable** CPUs. If there is a lot of processing by the client or other non-server processes, then one engine per CPU may be excessive. Remember, too, that the operating system may take up part of one of the CPUs.
- Have **enough** engines. It is good practice to start with few engines and add additional ones when the existing CPUs are almost fully used. If there are too few engines, the capacity of the existing engines will be exceeded and bottlenecks may result.

Monitoring CPU Usage

To maintain the correct number of engines, monitor CPU usage with an operating system utility. Consult the *System Administration Guide Supplement* for the appropriate utility for your operating system.

Managing Memory

The **memory** configuration variable may require special attention in SMP sites.

Not all platforms require a higher memory configuration variable than before. If your platform does, the installed value of the memory configuration variable reflects this, so you may never need to adjust it. If error message 701,

```
There is insufficient memory to run this query
```

appears frequently in the error log and at the client terminal, you may wish to increase the amount of memory available.

Managing Disks

The additional processing power of the SMP product may increase demands on the disks. Therefore, it is best to spread data across multiple devices for heavily used databases. Specify these devices when using the **create database** command. (See Chapter 14, ‘‘Managing Multiprocessor Servers’’, for a discussion of **create database**.)

Application Design Considerations

The SMP SQL Server is compatible with uniprocessor SQL Server. Applications that run on uniprocessor servers should run on SMP servers as well.

Concurrency

Because the increased throughput of the SMP SQL Server makes it more likely that multiple processes may try to access a data page simultaneously, it is especially important to follow principles of good database design in order to avoid contention. The following are some of the application design considerations that are especially important in an SMP environment.

Multiple Indexes

The increased throughput of SMP may result in increased lock contention when tables with multiple indexes are updated. Allow no more than two or three indexes on any table that will be updated often.

Adjusting the **fillfactor** *for* **create index** *Commands*

You may need to adjust the **fillfactor** in **create index** commands. Because of the added throughput with SMP, setting a lower **fillfactor** may reduce contention for the data pages.

Transaction Length

Transactions that include many statements or take a long time to run may result in increased lock contention. Keep transactions as short as possible.

Temporary Tables

Temporary tables (tables in *tempdb*) do not cause contention, because they are associated with individual users and are not shared.

15 Managing Remote Servers

Users on a local SQL Server can execute stored procedures on a remote SQL Server. Executing a **remote procedure call** sends the results of the remote process to the calling process—usually the user's screen. This chapter discusses the steps the System Administrator and System Security Officer of each SQL Server must execute to enable remote procedure calls. They are:

On the local server:

- Both the local server and remote server must be listed in the system table *master..sysservers* (with **sp_addserver**). This must be done by a System Security Officer.
- The remote server must be listed in the interfaces file for the local server. (The interfaces file, set up when SQL Server is installed, lists the names and addresses of all the servers you can access.)

On the remote server:

- The server originating the remote procedure call must be listed in the *sysservers* table (with **sp_addserver**). This is done by a System Security Officer.
- The user originating the remote procedure must be allowed access to the server (with **sp_addlogin** and **sp_addremotelogin**). **sp_addlogin** is executed by a System Security Officer, **sp_addremotelogin** by a System Administrator.
- The remote login name must be added as a user of the appropriate database and must have permission to execute the procedure. (If **execute** permission is granted to "public," the user does not need to be granted specific permission.) Figure 15-1 illustrates the steps.

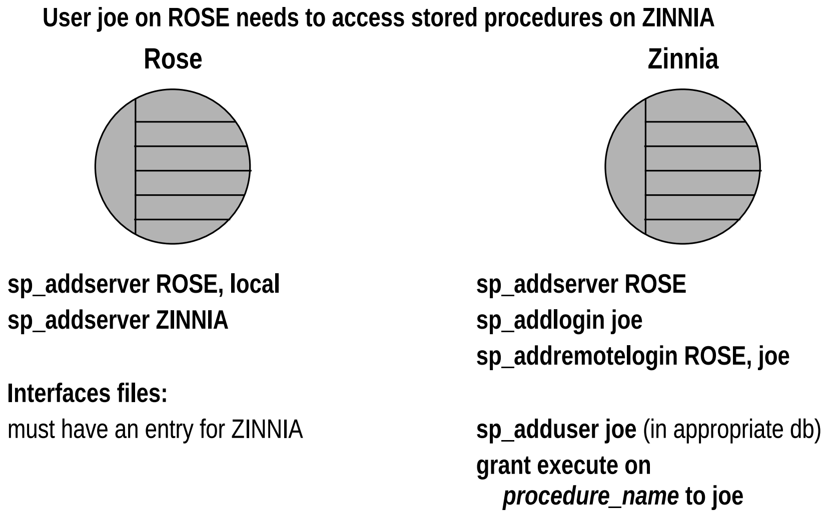


Figure 15-1: Setting Up Servers to Allow Remote Procedure Calls

Managing Remote Servers

Four system procedures are used to manage remote servers:

- **sp_addserver:** Adds a server name to *master..sysservers*
- **sp_dropserver:** Drops a server name from *master..sysservers*
- **sp_helpserver:** Displays information about servers
- **sp_serveroption:** Displays or changes server options

Adding a Remote Server

The system procedure **sp_addserver** adds entries to the *sysservers* table. On the server originating the call, you must add an entry for the local server, and an entry for each remote server that your server will call. Only a System Security Officer can execute **sp_addserver.** When you create entries for a remote server, you can choose to:

- Always refer to them by the name listed in the interfaces file, or
- Provide a local name for the remote server. For example, if the name in the interfaces file is "MAIN_PRODUCTION," you may want to call it simply "main."

Here is the syntax:

```
sp_addserver server_name [{, local | null}
    [, network_name]]
```

- The *server_name* parameter provides the local "call name" for the remote server. Users will type this name when they issue remote procedure calls. If this name is **not** the same as the remote server's name in the interfaces file, you must provide that name as the third parameter, *network_name*.

  The remote server must be listed in the interfaces file on the local machine. If it's not listed, copy the interfaces file entry from the remote server and append it to your existing interfaces file. Be sure to keep the same port numbers.
- The **local** option identifies the server being added as a local server. The **local** value is used only once after start-up, or after a reboot, to identify the local server name so that it can appear in messages printed out by SQL Server. **null** specifies that this server is a remote server.
- For *network_name*, give the name for the remote server that is listed in the interfaces file for the server named *srvname*. This optional third argument permits you to establish local aliases for other SQL Servers, Open Servers, or Backup Servers that you may need to communicate with. If you do not specify a *network_name*, it defaults to *srvname*.

The server originating a remote procedure call must have a local entry. You must restart SQL Server to set the value of the global variable *@@servername*.

The following examples create entries for the local server named DOCS:

```
sp_addserver DOCS, local
```

This example creates an entry for a remote server named GATEWAY:

```
sp_addserver GATEWAY
```

Here's how a user could run **sp_who** on GATEWAY:

```
GATEWAY.sybsytemprocs.dbo.sp_who
```

or

```
GATEWAY...sp_who
```

The following example gives a remote server called MAIN_PRODUCTION the local alias "main":

```
sp_addserver main, null, MAIN_PRODUCTION
```

Users can then type:

```
main...sp_who
```

Managing Remote Server Names

The *master.dbo.sysservers* table has two name columns:

- *srvname* is the name that users must supply when executing remote procedure calls. *srvname* must be unique on each server.
- *srvnetname* is the server's network name, which must match the name in the *interfaces* file. *srvnetname* does not have to be unique, you can have more than one local name for a remote server.

If you need to add or drop servers from your network, you can use **sp_addserver** to update the *srvnetname*. If you need to remove the server MAIN_PRODUCTION from the network, and move your remote applications to TEMP_PRODUCTION, here's how to change the network name, while keeping the local alias:

```
sp_addserver main, null, TEMP_PRODUCTION
```

The **sp_addserver** procedure prints an informational message to tell you that it is changing the network name of an existing server entry.

Dropping Remote Servers: *sp_dropserver*

The **sp_dropserver** system procedure drops servers from *sysservers*. The syntax is:

sp_dropserver ***server_name*** **[, droplogins]**

Only System Security Officers can execute **sp_dropserver.** The **droplogins** option allows you to drop a remote server and all of that server's remote login information in one step. If you don't use the **droplogins** option, you cannot drop a server which has remote logins associated with it. This statement drops the GATEWAY server, and all of the remote logins associated with it:

```
sp_dropserver GATEWAY, droplogins
```

The **droplogins** option isn't needed if you wish to drop the local server's entry; that entry won't have remote login information associated with it.

Setting Server Options: *sp_serveroption*

The **sp_serveroption** system procedure sets the server options **timeouts** and **net password encryption**, which affect connections with remote servers. These options do not affect SQL Server to Backup Server communication.

The timeouts *Option*

timeouts disables and enables the normal timeout code used by the local server, so the site connection handler does not automatically drop the physical connection after a minute with no logical connection. This option can be set only by a System Administrator.

By default, **timeouts** is set to TRUE, and the site handler process that manages remote logins times out if there has been no remote user activity for one minute. By setting timeouts to FALSE on both of the servers involved in remote procedure calls, the automatic timeout is disabled. This example changes the timeouts option to FALSE:

```
sp_serveroption GATEWAY, "timeouts", false
```

Once you set **timeouts** to FALSE on both servers and a user executes a remote procedure call in either direction, the site handler on each machine runs until one of the servers is shut down. (When the server is brought up again, the option remains FALSE, and the site handler will be re-established the next time a user executes a remote procedure call.) If users on the server execute remote procedure calls frequently, it's probably efficient in terms of system resources to set this option to FALSE, since there is some system overhead involved in setting up the physical connection.

The net password encryption *Option*

net password encryption specifies whether connections with a remote server are to be initiated with a client-side password encryption handshake or with the normal (unencrypted password) handshake sequence. The default is FALSE. This option can be set only by a System Security Officer.

If **net password encryption** is set to TRUE, the following occurs:

1. The initial login packet is sent without passwords.
2. The client indicates to the remote server that encryption is desired.
3. The remote server sends back an encryption key, which the client uses to encrypt its plaintext passwords.

4. The client then encrypts its own passwords, and the remote server uses the key to authenticate them when they arrive.

This example sets this option to TRUE:

```
sp_serveroption GATEWAY, "net password encryption",
    true
```

Setting this option has no effect on SQL Server's interaction with Backup Server.

Getting Information on Servers: *sp_helpserver*

The system procedure **sp_helpserver** reports on servers. When it is used without an argument, it provides information on all the servers listed in *sysservers*. It can take a server name as an argument, and provides information on just that server.

sp_helpserver [*server_name*]

sp_helpserver checks for both *srvname* and *srvnetname* in the *master..sysremotelogins* table.

Adding Remote Logins

The System Security Officer and System Administrator of any SQL Server share control over which remote users can access the server, and what identity the remote users will assume. The System Administrator adds remote logins with the **sp_addremotelogin** system procedure. The System Security Officer controls whether password checking will be required with the **sp_remoteoption** system procedure.

Mapping User's Server IDs: *sp_addremotelogin*

Logins from a remote server can be mapped to a local server in three ways:

- A particular remote login can be mapped to a particular local login name. For example, the user "joe" on the remote server might be mapped to "joesmith", user "judy" to "judyj", and so forth.
- All logins from one remote server can be mapped to one local name. For example, all users sending remote procedure calls from the MAIN_PRODUCTION server might be mapped to "remusers".
- All logins from one remote server can use their remote names.

The first option can be combined with the other two options, and its particular mapping takes precedence over the other two more general mappings. The second two options are mutually exclusive; you may use either, but not both of them.

The system procedure **sp_addremotelogin** adds remote logins. The syntax is:

```
sp_addremotelogin remoteserver [, localname
   [, remotename]]
```

Only a System Administrator can execute **sp_addremotelogin**. If the local names are not listed in *master..syslogins*, you must first add them as SQL Server logins with **sp_addlogin**.

Mapping Remote Logins to Particular Local Names

The following example maps the login named "pogo" from a remote system to the local login name "bob." The user logs into the remote system as "pogo"; whenever that user executes remote procedure calls from GATEWAY, the local system maps the remote login name to "bob."

```
sp_addlogin bob
sp_addremotelogin GATEWAY, bob, pogo
```

Mapping All Remote Logins to One Local Name

The following example creates an entry that maps all remote login names to the local name "albert." All names are mapped to "albert," except those with specific mapping, as described above. For example, if you mapped "pogo" to "bob," and then the rest of the logins to "albert," "pogo" still maps to "bob."

```
sp_addlogin albert
sp_addremotelogin GATEWAY, albert
```

◆ ***WARNING!***

Mapping more than one remote login to a single local login is not recommended, as it reduces individual accountability on the server. Audited actions can be traced only to the local server login, not to the individual logins on the remote server.

Remote Logins Keeping Remote Names

If you want remote users to keep their remote login names while using a local server, first use **sp_addlogin** to create a login for each login from the remote server. Then, use **sp_addremotelogin** for the server as a whole:

```
sp_addremotelogin GATEWAY
```

This command creates an entry in *master..sysremotelogins* with a null value for the remote login name, and value of -1 for the *suid*.

The option of mapping individual remote logins to different local logins takes precedence over the process of giving all remote logins the same name, so you can use some combination of these two options. However, you cannot combine the second two options; they are mutually exclusive. If you want to change from one method to the other, you must use **sp_dropremotelogin** to remove the old mapping.

An Example of Remote User Mapping

Assume that you have the following servers listed in *master..sysservers* on the SALES server:

```
select srvid, srvname from sysservers
```

```
srvid  srvname
-----  ----------
    0  SALES
    1  CORPORATE
    2  MARKETING
    3  PUBLICATIONS
    4  ENGINEERING
```

and the following entries in *master..sysremotelogins*:

```
remoteserverid  remoteusername  suid
--------------  --------------  ------
    1               joel            3
    1               freddy          3
    1               NULL            4
    3               NULL            5
    4               NULL            -1
```

and these entries in *master..syslogins*:

```
select suid, name from syslogins
```

```
suid    name
------  ------------
     1  sa
     2  probe
     3  vp
     4  peon
     5  writer
```

The effect is:

- The logins "joel" and "freddie" from the CORPORATE server are known as "vp."
- All other logins from CORPORATE are mapped to the name "peon."
- All logins from PUBLICATIONS are mapped to "writer."
- All logins from ENGINEERING are looked up in *master..syslogins* by their remote user names.
- Users from MARKETING cannot run remote procedure calls on this server, since there is no entry for MARKETING in *sysremotelogins*.

The remote user mapping procedures and the ability to set permissions for individual stored procedures give you control over which remote users can access local procedures. For example, the "vp" login from CORPORATE could execute certain local procedures, while all of the other logins from CORPORATE could only execute the procedures for which "peon" had permission.

➤ ***Note***

Note that in many cases, the passwords for "joel" and "freddie" on the corporate server must match the password for the SALES login name "vp," as explained in the next section.

Password Checking for Remote Users: *sp_remoteoption*

The system procedure **sp_remoteoption** determines whether passwords will be checked when remote users log into the local server. By default, passwords are verified ("untrusted" mode). In **trusted** mode, the local server accepts remote logins from other servers and front-end applications without user-access verification for the particular login.

◆ ***WARNING!***

Using the trusted mode of sp_remoteoption reduces the security of your server, as passwords from such "trusted" users are not verified.

When **sp_remoteoption** is used with arguments, it changes the mode for the named user. Here is the syntax:

```
sp_remoteoption [remoteserver, loginame, remotename,
   optname, {true | false}]
```

Only a System Security Officer can execute **sp_remoteoption.**

This example sets **trusted** mode for the user "bob":

```
sp_remoteoption GATEWAY, pogo, bob, trusted,
    true
```

The effects of the "untrusted" mode depend on the user's client program. **isql** and some user applications require that logins have the same password on the remote server and the local server. Open Client applications can be written to allow local logins to have different passwords on different servers.

In "untrusted" mode, if you want to change your password you must change it on all the remote systems you access before changing it on your local server.This is because of the password checking; if you change your password on the local server first, when you issue the remote procedure call to execute **sp_password** on the remote server your passwords will no longer match. The syntax for changing your password on the remote server is:

```
remote_server...sp_password old_passwd, new_passwd
```

And on the local server:

```
sp_password old_passwd, new_passwd
```

See ''Changing Passwords: sp_password'' on page 4-13 for more information about changing your password.

Getting Information about Remote Logins

The stored procedure **sp_helpremotelogin** prints information about the remote logins on a server. The following example shows the remote login "pogo" mapped locally to login name "bob," with all other remote logins keeping their remote names.

```
sp_helpremotelogin
```

```
server     remote_user_name    local_user_name     options
-------    ----------------    -----------------   --------
GATEWAY    **mapped locally**  **use local name**  untrusted
GATEWAY    pogo                bob                 untrusted
```

Configuration Variables for Remote Logins

Each SQL Server that allows remote logins to execute procedure calls must have certain configuration variables set. These values are changed with **sp_configure**, and the **reconfigure** command.

The values that affect remote procedure calls are: **remote access**, **remote logins**, **remote sites**, **remote connections**, and **pre-read packets**. By default, **remote access** is set to 1, and other values are set to reasonable defaults. You can also accept default values for the last four of these variables by resetting only **remote access**, executing **reconfigure**, and restarting the server. These are the default settings:

Configuration Variable	Default
remote access	1
remote logins	20
remote sites	10
remote connections	20
pre-read packets	3

Table 15-1: Configuration Variables that Affect RPCs

The **remote access** configuration variable can be changed only by a System Security Officer. The other variables can be changed only by a System Administrator. Only a System Administrator can execute the **reconfigure** command.

remote access

The **remote access** variable must be set to 1 to allow remote access to or from a server, including access to the Backup Server:

```
sp_configure "remote access", 1
```

If you wish to disallow remote access at any time, the **remote access** variable can be reset to 0:

```
sp_configure "remote access", 0
```

If you set **remote access** to 0, and then execute **reconfigure**, the run value of the other four remote-access variables is also set to 0.

If you don't use the default values, the values for the other remote access configuration variables will depend on your system design and applications.

Only a System Security Officer can set the **remote access** variable.

◆ ***WARNING!***

You cannot perform database or transaction log dumps while the remote access configuration variable is set to 0.

remote logins

The **remote logins** variable controls the number of active user connections from this site to remote servers. Only a System Administrator can set **remote logins**. This command increases **remote logins** to 50:

```
sp_configure "remote logins", 50
```

remote sites

The **remote sites** variable controls the number of simultaneous remote sites that can access this server. All accesses from an individual site are managed by one site handler; this variable controls the number of site handlers, not the number of individual simultaneous procedure calls. If you set **remote sites** to 5, and each site initiates three remote procedure calls, you will have 5 site handler processes listed (use **sp_who** to see them) for the 15 processes. Only a System Administrator can set **remote sites**.

remote connections

The **remote connections** variable controls the limit on active connections initiated to and from this server. This includes those initiated from this server and those from remote sites to this server. Only a System Administrator can set **remote connections**.

pre-read packets

All communication between two servers is handled through one site handler to reduce the needed number of connections. This site handler can pre-read and keep track of data packets for each user before the user process that needs them is ready. The **pre-read packets** variable controls how many packets a site handler will pre-read. The default value 3 is adequate in virtually all cases; higher values can use too much memory. Only a System Administrator can set **pre-read packets**.

16 Auditing

Introduction

Auditing is an important part of security in a database management system. Through it, security-related system activity is recorded in an audit trail, which can be used to detect penetration of the system and misuse of resources. By examining the audit trail System Security Officers can inspect patterns of access to objects in databases, and can monitor the activity of specific users. Audit records are traceable to specific users, enabling the audit system to act as a deterrent to users attempting to misuse the system.

In the sections that follow, the audit system is discussed in detail, including what is audited, how the system works, and how to interpret the audit trail.

The Audit System

The audit system consists of:

- The *sybsecurity* database
- System procedures that set the various auditing options
- The audit queue, to which audit records are sent before they are written to the audit trail

The *sybsecurity* Database

The *sybsecurity* database is essential to auditing in SQL Server. It is created as part of the auditing installation process. It contains all system tables found in the *model* database and two additional system tables:

- *sysaudits*: the audit trail
- *sysauditoptions*: contains the global auditing choices

The contents of the audit trail is discussed in detail in “The Audit Trail: sysaudits” on page 16-16.

The Auditing System Procedures

Auditing is managed with the following system procedures:

System Procedure	Description
sp_auditoption	Enables and disables system-wide auditing and global audit options.
sp_auditdatabase	Establishes auditing of different types of events within a database, or of references to objects within that database from another database.
sp_auditobject	Establishes selective auditing of accesses to tables and views.
sp_auditsproc	Audits the execution of stored procedures and triggers.
sp_auditlogin	Audits a user's attempts to access tables and views, or the text of commands that the user executes.
sp_addauditrecord	Allows users to enter user-defined audit records (comments) into the audit trail.

Table 16-1: System Procedures Used to Manage Auditing Options

The Audit Queue

When an audited event occurs, an audit record first goes to the in-memory audit queue, where it is held until it can be processed by the audit process and added to the audit trail. As long as an audit record is in the queue, it can be lost if the system crashes. However, if the queue repeatedly becomes full, overall system performance is affected: if the audit queue is full when a user process tries to generate an audit record, the process sleeps until space in the queue becomes available. Thus, there is a performance / risk trade-off that must be considered when configuring the queue size.

You can configure the size of the audit queue with the **audit queue size** option to **sp_configure**. See "audit queue size" on page 12-37 for information on how to configure the size of the audit queue and effects of different queue sizes.

Installing the Audit System

The audit system is installed with **sybinit**, the Sybase installation program. Installation and **sybinit** are discussed in the *SYBASE Installation Guide* for your platform.

sybinit installs the *sybsecurity* database on its own device and the *sysaudits* table on its own segment on that device. This allows you to use the threshold manager to monitor the available space in *sysaudits*. See "Archiving Audit Data" on page 16-19 for more information.

◆ ***WARNING!***

Do not put any objects into the *sybsecurity* database besides those that are automatically installed there. Because *sysaudits* is so dynamic, you must carefully monitor the space on its device. This is harder to do if there are extra objects on the device.

Because it is assumed that there will be no extra activity in the *sybsecurity* database, the **trunc log on chkpt** database option is turned on automatically when *sybsecurity* is installed. This means that the log in that database is truncated every time the **checkpoint** process occurs, and the transaction log cannot be dumped or recovered. Inserts into the *sysaudits* table are not logged.

Removing the Audit System

If you need to remove the audit system and reclaim the disk space, follow this procedure:

1. Disable any current auditing with this command:

   ```
   sp_auditoption "enable auditing", "off"
   ```

2. Remove the audit database with this command:

   ```
   drop database sybsecurity
   ```

You cannot remove *sybsecurity* if auditing is enabled. Only a System Security Officer can drop *sybsecurity*.

If at a later time you wish to reinstall the audit system, run **sybinit** again and access the menus that install auditing.

Establishing Auditing

The System Security Officer manages the audit system. Only a user who has been granted that role can:

- Execute any of the auditing system procedures (with the exception of **sp_addauditrecord**)

- Read the audit trail
- Access the audit database

The System Security Officer who is going to manage the audit system must also be granted access to the *sybsecurity* database.

Turning Auditing On and Off

Once you have installed the *sybsecurity* database and the auditing system procedures, you can set all of the audit options as you wish. No auditing actually occurs, however, until you turn auditing on for the server as a whole using this command:

```
sp_auditoption "enable auditing", "on"
```

Turn server-wide auditing off with this command:

```
sp_auditoption "enable auditing", "off"
```

Turning off this option does not modify your auditing set-up, though all auditing stops when **enable auditing** is set to **off**.

sp_auditoption is discussed in the following section.

Setting the Global Audit Options: *sp_auditoption*

The global audit options are those that affect the server as a whole. They are enabled and disabled with **sp_auditoption**, and information about their current settings is contained in the *sybsecurity..sysauditoptions* system table.The available options and their syntax are described in Table 16-2.

Option	Action
enable auditing	Enables or disables system-wide auditing. A System Security Officer must set the **enable auditing** option to **on** before any other auditing can take place. Enabling or disabling auditing automatically generates an audit record, so that you can bracket time periods when auditing was enabled. Syntax: `sp_auditoption "enable auditing" [, {"on"|"off"}]`
all	Enables or disables all options **except enable auditing** simultaneously. **enable auditing** must be set separately. For options that allow selective auditing for successful and / or failed executions, setting **all** to **on** is equivalent to setting all options to **on** or **both**, depending on the option. Syntax: `sp_auditoption "all" [, {"on"|"off"}]`
logins	Enables or disables auditing of successful (**ok**), failed (**fail**), or all (**both**) login attempts by all users. To audit individual users, use **sp_auditlogin**. Syntax: `sp_auditoption "logins" [, {"ok"|"fail"|"both"|"off"}]`
logouts	Enables or disables auditing of all logouts from the server, including unintentional logouts such as dropped connections. Syntax: `sp_auditoption "logouts" [, {"on"|"off"}]`
server boots	Enables or disables generation of an audit record when the server is rebooted. Syntax: `sp_auditoption "server boots" [, {"on"|"off"}]`
rpc connections	When this option is **on**, it generates an audit record whenever a user from another host connects to the local server to run a procedure via a remote procedure call (RPC). Auditing can be enabled for all connection attempts (**both**), successful attempts only (**ok**), or failed attempts only (**fail**). Syntax: `sp_auditoption "rpc connections" [, {"ok"|"fail"| "both"|off"}]`
roles	Audits the use of the **set role** command to turn roles on and off. You can enable auditing of all attempts (**both**), successful attempts only (**ok**), or failed attempts only (**fail**). See Chapter 2, ‘‘Roles in SQL Server,’’ for more information. Syntax: `sp_auditoption "roles" [, {"ok"|"fail"|"both"|"off"}]`
{**sa** | **sso** | **oper** | **navigator** | **replication**} **commands**	Audits the use of privileged commands—those requiring one of the roles for execution. You can enable auditing for successful executions only, failed attempts (where failure is due to the user lacking the proper role), or both. See “Roles” in the *SQL Server Reference Manual* for a list of the commands that require the various roles. Syntax: `sp_auditoption "{sa|sso|oper|navigator|replication} commands" [, {"ok"|"fail"|"both"|"off"}]`

Table 16-2: Global Auditing Options

Option	Action
errors	Audits fatal errors (errors that break the user's connection to the server and require the client program to be restarted), nonfatal errors, or both kinds of errors. Fatal errors do not include server internal fatal software errors (bus errors, segmentation faults, etc.) In case of internal errors, information will be contained in the errorlog file for the server. Syntax: `sp_auditoption "errors" [, {"nonfatal"|"fatal" |"both"|"off"}]`
adhoc records	Allows users to send text to the audit trail via the **sp_addauditrecord** command. Syntax: `sp_auditoption "adhoc records", {"on"|"off"}`

Table 16-2: Global Auditing Options (continued)

The initial, default value of all options is "off."

Examples

Using **sp_auditoption** with no parameters, like this:

```
sp_auditoption
```

or with the **all** option, like this:

```
sp_auditoption "all"
```

displays the current settings for all of the global audit options.

Using **sp_auditoption** with the name of any option and no other parameters displays the current setting for that option. For example:

```
sp_auditoption "logins"
```

displays the current status of the **logins** option.

This command:

```
sp_auditoption "errors", "fatal"
```

establishes auditing of fatal errors.

This command:

```
sp_auditoption "logins", "both"
```

establishes auditing of all attempts to log into SQL Server, both successful and unsuccessful. To audit individual users, use **sp_auditlogin**.

Auditing Users: *sp_auditlogin*

sp_auditlogin allows you to audit a SQL Server user's attempts to access tables and views in any database, and the text of commands that the user sends to SQL Server.

Using **sp_auditlogin** with no parameters, like this:

```
sp_auditlogin
```

displays the audit status for all login names in the server.

Using **sp_auditlogin** with a login name and no other parameters, like this:

```
sp_auditlogin bob
```

displays the audit status for that user.

Auditing a User's Table and View Accesses

The syntax for auditing a user's attempts to access tables and views is:

```
sp_auditlogin [login_name [, "table" | "view"
   [, "ok"|"fail"|"both"|"off"]]]
```

An **access** is the use of the **select, insert, update,** or **delete** command on a table or view. The **table** option audits *login_name*'s attempts to access tables in any database. **view** audits *login_name*'s attempts to access views in any database. If you use **table** or **view** without the **ok | fail | both | off** parameter, **sp_auditlogin** displays the status of table or view auditing for the named user.

The **ok** parameter enables auditing for successful table or view accesses only. **fail** establishes auditing of accesses that fail because the user lacks permissions on an object. **both** establishes auditing of both successful and failed accesses. **off** disables auditing of the named type (**table** or **view**).

For example, this command:

```
sp_auditlogin "joe", "table", "both"
```

audits all of Joe's attempts to access any table on the server. This command:

```
sp_auditlogin joe, "table", "fail"
```

audits Joe's attempts to access tables on which he has not been granted permission.

This command:

```
sp_auditlogin bob, "table"
```

displays the status of table auditing for Joe.

Auditing the Text of a User's Commands

The syntax for auditing the text of commands that a user sends to SQL Server:

```
sp_auditlogin [login_name [, "cmdtext"
   [, "on"|"off"]]]
```

Set **cmdtext on** to write the text of a user's SQL Server commands to the audit trail.

Using the **cmdtext** option without the final parameter, like this:

```
sp_auditlogin bob, "cmdtext"
```

displays the status of **cmdtext** auditing for Bob.

Auditing Databases: *sp_auditdatabase*

The **sp_auditdatabase** system procedure permits you to establish selective auditing on databases. These are the auditable events that can occur in or on a database:

- Use of the **drop**, **grant**, **revoke**, and **truncate table** commands within a database
- Use of the **drop database** and **use** commands on a database
- Execution of SQL commands from within another database that reference the audited database

sp_auditdatabase Syntax

The syntax of **sp_auditdatabase** is:

```
sp_auditdatabase [dbname [, {"ok"|"fail"|"both"|"off"}
   [, "d u g r t o"}]]]
```

- If you use **sp_auditdatabase** with no parameters, it displays the current audit status for all databases in the server.
- *dbname* is the name of the database for which to establish auditing. If you use **sp_auditdatabase** with *dbname* and no other arguments, it displays the audit settings for the specified database.

- **ok** establishes auditing of successful executions of the commands specified in the third parameter. **fail** audits access attempts that fail because the user lacks permission to access the database. **both** audits successful and failed executions. **off** turns off the specified type of auditing for the named database.
- **d, u, g, r, t,** and **o** are the types of database events that you can audit. They are:

Event Type	Meaning
d	Audits execution of the **drop table**, **drop view**, **drop procedure**, or **drop trigger** commands within *dbname*, and execution of the **drop database** command when *dbname* is being dropped.
u	Audits execution of the **use** command on *dbname*.
g	Audits execution of the **grant** command within *dbname*.
r	Audits execution of the **revoke** command within *dbname*.
t	Audits execution of the **truncate table** command within *dbname*.
o	"Outside access"; audits execution of SQL commands from within another database that refer to objects in *dbname*.

Table 16-3: Database Auditing Options

Choose one or more, in any order. If none are specified, the **ok | fail | both | off** argument applies to all event types.

For example, this command:

```
sp_auditdatabase pubs2, "ok", "du"
```

enables auditing of successful executions of the **drop** and **use** commands on *pubs2*.

You can execute **sp_auditdatabase** more than once on a database, and the options that you set will accumulate with each execution. Therefore, you can set some options for success only, others for failure only, and still others for both success and failure. For example, this sequence:

```
sp_auditdatabase pubs2, "ok", "d"
go
sp_auditdatabase pubs2, "fail", "u"
go
sp_auditdatabase pubs2, "both", "gr"
go
```

establishes auditing on *pubs2* of successful **drop** commands, failed **use** commands, and all **grant** and **revoke** commands.

If, later, you no longer wish to audit execution of the **drop** command, issue this statement:

```
sp_auditdatabase pubs2, "off", "d"
```

If you wish to turn off all auditing of *pubs2*, use this command:

```
sp_auditdatabase pubs2, "off"
```

Auditing Tables and Views: *sp_auditobject*

sp_auditobject allows you to audit accesses of specified tables and views. An **access** is the use of the **select, insert, update,** or **delete** command on a table or view. You can establish auditing of specified tables and views, or create default audit settings for newly-created tables and views.

To establish auditing of an existing table or view, the syntax is:

```
sp_auditobject [owner.]objname, dbname
   [, "ok"|"fail"|"both"|"off" [,"{d u s i}"]]
```

To establish default audit settings for future tables and views in the current database, the syntax is:

```
sp_auditobject {"default table"|"default view"},
   dbname [, "ok"|"fail"|"both"|"off"
   [, "{d u s i}"]]
```

- If you are establishing auditing for an existing table or view, use the *objname* parameter. *objname* is the name of an existing table or view in the current database. If you are not the owner of the object, you must qualify its name with the name of the owner, like this:

  "ownername.objname"

- If you are establishing default audit settings for future tables or views in the current database, use the **default table | default view** parameter. This specifies that these audit settings are to be the defaults for any tables and views that will be created in the future. These default settings are not applied to any tables or views that exist at the time that **sp_auditobject** is executed. If you do not set any default audit settings in a database, newly-created tables and views will have no auditing options set initially.

- If you are using the *objname* parameter, *dbname* must be the name of the current database. You must be in that database to execute **sp_auditobject**, and you must also supply its name.

  If you are using the **default table** | **default view** option, *dbname* can be the name of any database.

- **ok** establishes auditing for accesses that are successful. **fail** audits attempts to access the object that fail because the user has not been granted permission on the object. **both** audits both types of access attempts. **off** turns auditing off for the named table or view.

- The **d u s i** parameter indicates what kinds of access to audit. They can be specified in any order, and you can specify any number of them at a time. If you omit these parameters, all four event types will be effected by the **ok | fail | both | off** parameter.The types of access are:

Parameter	Meaning
d	delete
i	insert
s	select
u	update

Table 16-4: Types of Object Access

Choose one or more, in any order. If none are specified, the **ok** | **fail** | **both** | **off** argument applies to all four of the event types.

Auditing Indirect References to Objects

If the named table or view is referenced **indirectly**, such as through a view or stored procedure, an audit record for that table or view is generated only if a permission check is required. The rules governing permission checks are as follows:

- If the **owner** of the stored procedure or view is also the owner of the underlying table or view, the **user** of the stored procedure or view is not required to have permissions on the underlying table or view. No permission checking is performed on the underlying object. Therefore, no audit record is generated when someone uses this view or stored procedure, even though auditing is enabled for the underlying table or view.
- If the **creator** of the stored procedure or view does **not** own the underlying table or view, the **user** of the stored procedure or view must have access permissions on the underlying table or view.

Permission checking is performed on the underlying object, and an audit record is generated if that table or view is configured to be audited.

See "Ownership Chains" on page 5-24 for information about ownership chains.

Examples

This command:

```
sp_auditobject authors, pubs2, "both", "dui"
```

establishes auditing on the *authors* table in the *pubs2* database. Both successful and failed attempts to execute the **delete, update**, and **insert** commands on *authors* will be audited.

You can execute **sp_auditobject** more than once on a table or view, and the options that you set will accumulate with each execution. Therefore, you can set some options for success only, others for failure only, and still others for both success and failure. For example, this sequence:

```
sp_auditobject titles, pubs2, "ok", "id"
go
sp_auditobject titles, pubs2, "fail", "u"
go
```

establishes auditing on the *titles* table so that successful executions of the **insert** and **delete** commands will be audited, and failed attempts to execute **update** commands will be audited.

This command:

```
sp_auditobject titles, pubs2, "off"
```

turns off all auditing on the *titles* table.

This command:

```
sp_auditobject publishers
```

displays the current audit settings for the *publishers* table.

This command:

```
sp_auditobject "default table", pubs2, "fail", "du"
```

establishes auditing of failed attempts to execute the **delete** and **update** commands on all new tables in the *pubs2* database. Any table created after this command is executed will automatically have these audit settings.

This command:

```
sp_auditobject "default view", pubs2
```

displays the default audit settings for views in the *pubs2* database.

Auditing Stored Procedures: *sp_auditsproc*

The **sp_auditsproc** system procedure allows you to audit the execution of stored procedures and triggers. The values of any parameters passed to a stored procedure are also audited. You can establish auditing of existing stored procedures or triggers, or create default audit settings for future stored procedures and triggers.

The syntax to establish auditing of existing stored procedures and triggers is:

```
sp_auditsproc {sproc_name | "all"},
   dbname [, {"ok"|"fail"|"both"|"off"}]
```

The syntax to establish default audit settings for future stored procedures and triggers in the current database is:

```
sp_auditsproc "default", dbname
   [, {"ok"|"fail"|"both"|"off"}]
```

- If you are establishing auditing for an existing stored procedure or trigger, use the *sproc_name* parameter. *sproc_name* is the name of the stored procedure or trigger on which you want to establish auditing. If you are not the owner of the object, you must qualify its name with the owner's name, like this:

  ```
  "ownername.sproc_name"
  ```

  If you wish to establish auditing for all existing triggers and stored procedures in the current database, specify **all**.

- If you are establishing default audit settings for future stored procedures and triggers in the current database, use the **default** parameter. This specifies that these audit settings are to be the defaults for any stored procedures and triggers that will be created in the future. These default settings are not applied to any stored procedures or triggers that exist at the time that **sp_auditsproc** is executed. If you do not set any default audit settings in a database, newly-created stored procedures and triggers will not be audited.

- If you are using the *sproc_name* | **all** parameter, *dbname* must be the name of the current database. You must be in this database to execute **sp_auditobject**, and you must also supply its name.

If you are using the **default** parameter, *dbname* can be the name of any database.

- **ok** establishes auditing for executions of the named stored procedure or trigger that are successful. **fail** audits attempts to execute the procedure that fail because the user has not been granted permission on the stored procedure, or because the user does not have permission on the underlying table or view. (Permissions are not granted on triggers, so this option does not apply to them.) **both** audits both types of execution attempts. **off** turns auditing off for the named stored procedure or trigger.

Examples

Used with no parameters, like this:

```
sp_auditsproc
```

sp_auditsproc displays the auditing status of all stored procedures and triggers in the current database.

Used with the name of a stored procedure or trigger and no other parameters, like this:

```
sp_auditsproc new_sproc
```

sp_auditsproc displays the audit status for *new_sproc*.

Used with the **default** parameter and no others, like this:

```
sp_auditsproc default
```

sp_auditsproc displays the default audit settings for stored procedures and triggers in the current database.

This command:

```
sp_auditsproc "all", pubs2, "fail"
```

establishes auditing of failed executions of all stored procedures and triggers in the *pubs2* database.

This command:

```
sp_auditsproc "default", pubs2, "ok"
```

sets a default in the *pubs2* database so that successful executions of new stored procedures and triggers will be audited.

Adding User-Specified Records to the Audit Trail

sp_addauditrecord allows users to enter user-defined audit records (comments) into the audit trail. The syntax is:

```
sp_addauditrecord [@text="message text"]
   [, @db_name="db_name"] [, @obj_name="object name"]
   [, @owner_name="object owner"]
   [, @dbid=database ID] [, @objid=object ID]
```

All of the parameters are optional.

- **@text** – is the text of the message that you wish to add to *sysaudits*. The text is inserted into the *extrainfo* field of *sysaudits*.
- **@db_name** – is the name of the database referred to in the record. This is inserted into the *dbname* field of *sysaudits*.
- **@obj_name** – is the name of the object referred to in the record. This is inserted into the *objname* field of *sysaudits*.
- **@owner_name** – is the owner of the object referred to in the record. This is inserted into the *objowner* field of *sysaudits*.
- **@dbid** – is the database ID number of *db_name*. This is an integer value, and must not be placed in quotes. **@dbid** is inserted into the *dbid* field of *sysaudits*.
- **@objid** – is the object ID number of *obj_name*. This is an integer value, and must not be placed in quotes. **@objid** is inserted into the *objid* field of *sysaudits*.

You can use **sp_addauditrecord** if:

- You have execute permission on **sp_addauditrecord**. (No special role is required.)
- Auditing is enabled (**sp_auditoption "enable auditing"** is set to **on**).
- The **adhoc records** option of **sp_auditoption** is set to **on.**

sp_addauditrecord does not check the correctness of the information you enter. For example, it does not check to see if the database ID you have entered is correct for the database name that you specify.

When the audit system is installed, only a System Security Officer and the Database Owner of *sybsecurity* can use **sp_addauditrecord**. Permission to execute it may be granted to other users.

Examples

The following example adds a record to *sysaudits*. The message portion is entered into the *extrainfo* field of *sysaudits*, "corporate" into the *dbname* field, “payroll” into the *objname* field, “dbo” into the *objowner* field, “10” into the *dbid* field, and “1004738270” into the *objid* field:

```
sp_addauditrecord @text="I gave A. Smith
permission to view the payroll table in the
corporate database. This permission was in effect
from 3:10 to 3:30 pm on 9/22/92.",
@db_name="corporate", @obj_name="payroll",
@owner_name="dbo", @dbid=10, @objid=1004738270
```

The following example inserts information only into the *extrainfo* and *dbname* fields of *sysaudits*:

```
sp_addauditrecord @text="I am disabling auditing
briefly while we reconfigure the system",
@db_name="corporate"
```

The Audit Trail: *sysaudits*

sysaudits is a special system table that exists only in the *sybsecurity* database and contains the audit trail. The only operations allowed on it are **select** and **truncate** commands, and these may be executed only by a System Security Officer.

Following are the columns of *sysaudits*:

- *event*: This contains a number corresponding to the event that is being audited. Table 16-5 describes the events and the contents of the *extrainfo* column that correspond to each event.

***event* No.**	**Description**	**Contents of *extrainfo* Column**
1	Enable auditing	NULL
2	Disable auditing	NULL
3	Login	Host name
4	Logout	Host name
5	Server boot	Names of the server program, master device, interfaces file path, server, and error log file
6	RPC connection	Remote server name, host name
7	Use of **set** command to turn roles on and off	Role, new setting
8	Command requiring **sa_role** role	Command type
9	Command requiring **sso_role** role	Command type
10	Command requiring **oper_role** role	Command type
12	Command requiring **navigator** role	Command type

Table 16-5: Contents and Description of event and extrainfo Columns

event No.	Description	Contents of *extrainfo* Column
13	Error	Error number, severity, and state
14	Ad hoc audit record	User-supplied comment text
15	Command requiring **replication** role	Command type
100	Database reference	Command type
101	Table reference	Command type
102	View reference	Command type
103	Stored procedure execution	Parameter list
104	Trigger execution	NULL
105	User's attempts to access a table	Command type
106	User's attempt to access a view	Command type
107	User's command text auditing	Command batch text

Table 16-5: Contents and Description of event and extrainfo Columns (continued)

- *eventmod* (modifier): This supplements the *event* field by further categorizing the type of event being audited. For example, this field will contain a different value for a successful login than for a failed login attempt.

  Possible values for the *eventmod* field are:

Value in *eventmod* Field	Description
0	No *eventmod* for this audited event.
1	Successful occurrence of this audited event. For error auditing (*event* code 12), a nonfatal error.
2	Failed occurrence of this audited event. For error auditing (*event* code 12), a fatal error.

Table 16-6: Values for the eventmod Field in sysaudits

- *spid* (server process ID): The ID number of the server process that caused the audit record to be written.
- *eventtime*: The date and time that the audited event occurred.
- *sequence*: The sequence number of the record within a single event (some events generate more than one audit record).
- *suid*: The server user ID of the user who performed the audited function.

- *dbid*: Depending on the type of event being audited, this is the ID of the database in which:
  - The object, stored procedure, or trigger resides, or
  - The audited event occurred.
- *objid*: The ID of the object accessed by the audited event. When a stored procedure or trigger is being audited, this field contains the ID of the relevant stored procedure or trigger.
- *xactid*: The transaction ID of the transaction containing the audited event. This can be useful for associating the audit record to a specific transaction in the transaction log. For a multi-database transaction, this field contains the transaction ID from the database where the transaction originated.
- *loginname*: The login name of the user who performed the audited function.
- *dbname*: The name of the database corresponding to *dbid*.
- *objname*: The name of the object corresponding to *objid*.
- *objowner*: The name of the owner of the object accessed by the audited event.
- *extrainfo*: Additional information regarding the audited event. See Table 16-5 on page 16-16 for a description of the contents of *extrainfo* for each event.

Reading the Audit Trail

The *sysaudits* table can be accessed only by a System Security Officer, who can read the audit record by executing SQL commands on it. The only commands that are allowed on *sysaudits* are **select** and **truncate**.

For example, this command requests audit records for tasks performed by “bob” on July 5, 1993:

```
select * from sysaudits
where loginname = "bob"
and eventtime like "Jul 5% 93"
```

This command requests audit records for commands performed on or in the *pubs2* database by users with the System Security Officer role (that kind of role auditing is *event* type 9, as shown in Table 16-5 on page 16-16):

```
select * from sysaudits
where event = 9
and dbname = "pubs2"
```

Archiving Audit Data

Because the *sysaudits* table is added to continuously, it is necessary to archive old audit data from time to time, depending on the size of the device on which *sysaudits* resides. If the audit device fills up, audited system activity will come to a halt.

To archive audit data, you have two choices:

- Use the **select into** command to create a new table and copy data into it.
- Use the **insert** and **select** commands to copy data from *sysaudits* into a pre-existing table.

The archive table should be in a separate database on a separate device from the *sybsecurity* database. It is preferable that this database be a special archive database, where you can preserve the audit records for as long as you need to.

The archive procedures are described in the following sections.

Using *select into*

The **select into** command creates a new table based on the table from which you are selecting. Before you can use **select into**, you must turn on the **select into/bulkcopy** database option in the archive database (with **sp_dboption**). Use the following procedure to copy *sysaudits*:

1. Create the archive database on a separate device from the one containing *sysaudits*.
2. In that database, use **sp_dboption** to turn on the **select into/bulk copy** option.
3. Copy the contents of *sysaudits* into this database with a command like this:

   ```
   select * into auditarchive7_5
   from sybsecurity.dbo.sysaudits
   ```

   You can further qualify the rows to be selected from *sysaudits* if you wish to.
4. Delete all rows from *sysaudits* with this command:

```
truncate table sysaudits
```

While the **select into/bulkcopy** option is on, you are not allowed to dump the transaction log, because these operations are not logged and changes are therefore not recoverable from transaction logs. In this situation, issuing the **dump transaction** statement produces an error message instructing you to use **dump database** instead.

Be certain that you dump your database before you turn off the **select into/bulkcopy** flag.

See **select** in Volume 1 of the *SQL Server Reference Manual* for more information.

Using *insert* and *select* to Copy Into an Archive Table

You can use this method to copy the audit data into an existing table having the same columns as *sysaudits*.

The procedure is:

1. Create the archive database on a separate device from the one containing *sysaudits*.
2. Create an archive table with columns identical to those in *sysaudits*. If such a table does not already exist, you can use **select into** to create an empty one by having a false condition in the **where** clause. For example:

   ```
   select *
   into auditarchive7_5
   from sybsecurity.dbo.sysaudits
   where 1 = 2
   ```

   The **where** condition is always false, so an empty table is created that is duplicate of *sysaudits*.

   The **select into/bulk copy** database option must be turned on in the archive database (using **sp_dboption**) before you can use **select into**.
3. Insert all rows from *sysaudits* into the archive table, like this:

   ```
   insert auditarchive7_5
   select * from sybsecurity.dbo.sysaudits
   ```
4. Delete all rows from *sysaudits* with this command:

   ```
   truncate table sysaudits
   ```

Using a Threshold Action Procedure

sysaudits is automatically installed on its own segment on the audit device. Therefore, you can use the **sp_addthreshold** system procedure to specify a user-supplied "threshold action" procedure to be executed when the free space left on the *sysaudits* segment goes below a specified threshold. (Threshold management is discussed in Chapter 10, "Managing Free Space with Thresholds".)

The "threshold action" stored procedure should perform at least these two tasks:

1. Execute a **select** or **insert** with **select** statement from *sysaudits* to an archive table, preferably in a separate archive database.
2. Truncate the *sysaudits* table with the **truncate table** command.

You can then manage the archived audit data with **dump** and **load** commands as desired.

Recovering if *sysaudits* Fills Up

If you do not regularly archive your audit data, the *sysaudits* table may run out of space. This, in turn, causes the audit device to become full. This is what happens next:

1. The audit process attempts to insert the next audit record into *sysaudits*. This fails, so the audit process dies. An error message goes to the error log.
2. As each user attempts to perform an auditable event (an event on which auditing has been enabled), the event cannot be completed because no auditing occurs. Each such user process dies. Users who do not attempt to perform an auditable event are unaffected.
3. If you have global login auditing enabled (with **sp_auditoption "logins"**), no one can log into the server, including a System Security Officer.
4. If you are auditing commands requiring System Security Officer privileges (with **sp_auditoption "sso_role", "on"**), a System Security Officer will be unable to execute any SSO-specific commands, such as to turn auditing off.

The procedure for recovering from this is as follows:

If the *sysaudits* device becomes full or the audit process terminates abnormally for any reason, the System Security Officer will immediately gain these special privileges:

- The System Security Officer becomes exempt from auditing. Every auditable event performed by a System Security Officer after this point will cause a warning message to be sent to the error log file. The message will state the date and time and a warning that an audit has been missed, as well as the user's server user ID, login name, *event* code, and any information that would go into the *extrainfo* field.
- The System Security Officer can execute the **shutdown** command to shut down the server after *sysaudits* has been archived and truncated. Normally, only the System Administrator can execute **shutdown**.

If *sysaudits* is full, the System Security Officer can now archive and truncate *sysaudits* as described in ''Archiving Audit Data'' on page 16-19. He or she can then execute **shutdown** to stop the server. A System Administrator can then restart the server, which will reestablish auditing.

If the audit process fails for a reason other than *sysaudits* becoming full, please call Sybase Technical Support.

17 Language, Sort Order, and Character Set Issues

Introduction

SQL Server supports international installations with a series of language modules that provide a variety of languages, sort orders and character sets for use by SQL Server. This chapter describes the language modules briefly, and summarizes the steps needed to change SQL Server's language, sort order, or character set.

International Language Modules

The international features of the SQL Server are supported by files that contain translated error messages, character set and sort order definitions, and utilities for converting SQL Server's character set into the appropriate character set for a particular terminal. These files, located in the *charsets* and *locales* directories in the *SYBASE* directory structure, are called **localization files**. and they are included when any language module is installed with your SQL Server. A U.S. English-only installation that does not receive these language modules does have some localization files, which are discussed in this chapter.

Types of Localization Files

Several different kinds of localization files are supplied with SQL Server.

File	Location	Purpose/Contents
productname.loc	In the character set subdirectories, under each language directory in the *locales* directory	Error messages translated into the local language. If an entry is not translated, that error message or string appears in U.S. English instead of the local language.
common.loc	In each language and character set directory of the *locales* directory	Contains the local names of the months of the year and their abbreviations, and information on the local date, time, and money formatting.

Table 17-1: Localization Files

File	Location	Purpose/Contents
charset.loc	In each character set subdirectory of the *charsets* directory	Character set definition files
**.srt*	In each character set subdirectory of the *charsets* directory	Defines the sort order for alpha-numeric and special characters, including ligatures, diacritics, and other language-specific considerations.
**.xlt*	Subdirectories under *charsets*	Terminal-specific character translations files for use with utilities such as **bcp** and **isql**.These files are part of the language module for Open Client. For more information about how the *.xlt* files are used, see the *SQL Server Utility Programs* or Chapter 18, ‘‘Converting Character Sets Between SQL Server and Client’’.

Table 17-1: Localization Files (continued)

◆ ***WARNING!***

Do not alter any of the localization files.If you need to install a new terminal definition or sort order, or if you want to alter an error message, contact your local SQL Server distributor.

The next section describes the directory structure of a language module.

Language Module Directory Structure

Figure 17-1: Language Module Directory Structure shows how language module files are arranged under the *charsets* and *locales* directories.This example shows character sets that are included in a Western European language module. Language modules for different regions may include different character sets and different types of *.srt* files. Supported languages and character sets in any of the language modules you receive are listed by the installation program when you are asked to select the languages and character sets you wish to install on your SQL Server. The sort orders usually available are described in the *SQL Server Installation Guide* and the *System Administration Guide Supplement*.

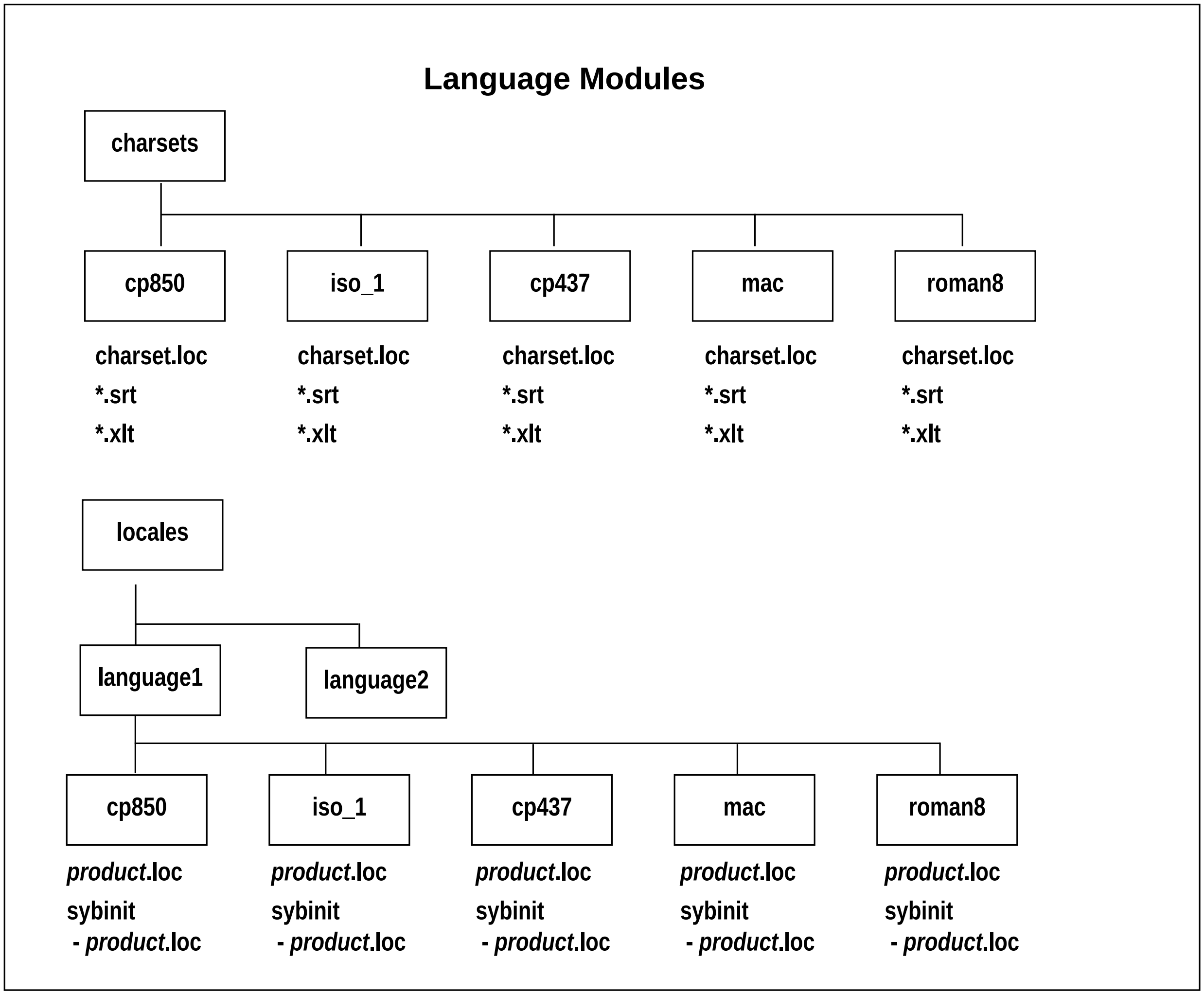


Figure 17-1: Language Module Directory Structure

Changing Sort Orders, Languages, and Character Sets

A System Administrator can change the sort order, language, or character set used by SQL Server. Because a sort order is built on a specific character set, changing character sets always involves a change in sort order. However, you can change sort orders without changing character sets, because there may be more than one sort order available for a character set. This section summarizes the steps to take when changing SQL Server's sort order, language, or character set using **sybinit**. See the *System Administration Guide Supplement* for your platform for more information on using this option.

About Changing Character Sets

The "Configure an existing SQL Server" option of **sybinit** is primarily intended to change character sets for those installations that are currently storing only 7-bit ASCII character data in their SQL Server databases. (Non-7-bit ASCII data includes 8-bit character data or multibyte data.) If you have non-7-bit ASCII data in your databases and you wish to change your character set, contact Sybase Technical Support.

◆ ***WARNING!***

It is safe to change your SQL Server's character set using sybinit only if all of your current character data is 7-bit ASCII.When you change character sets, existing character data is not converted to the new character set. As a result, changing character sets when you have non-7-bit ASCII data:

- **May create partial or illegal characters**
- **Changes the appearance of character data because different character sets encode characters in different ways**

About Changing Sort Orders

When you consider changing the sort order for character data on a particular SQL Server, keep this in mind: all your organization's SQL Servers should have the same sort order. A single sort order enforces consistency and distributed processing is easier to administer. If different SQL Servers support different non-binary sort orders, you cannot load a database dump from one to another. You can, however, load a database dump from one SQL Server to another SQL Server with a different character set if both servers use binary sort order. However, the character data that is loaded is interpreted as though it is in the new server's character set. No character set conversion is done.

SQL Server's sort order and character set are first determined when the server is installed, before you load any user data or database objects into it.

The Steps Involved

There are several steps involved in changing sort orders, languages, or character sets.

Preliminary Steps

1. Dump all user databases and *master*. If you have made changes to *model* or *sybsystemprocs*, dump it also.
2. Use **sybload** to load the language module if it is not already loaded (see the *SQL Server Installation Guide* for your platform for complete directions).
3. Move to the *install* directory of your SQL Server installation and invoke **sybinit**.
4. At the first prompt, select:

   `3. Configure a Server product`
5. Then, choose:

   `1. SQL Server`
6. And:

   `2. Configure an existing SQL Server`
7. Choose the number of the SQL Server from the list of server names.
8. Complete the password screen.

At this point, you can choose to configure languages, character sets, or sort order, as well as other options. As you complete each of these steps, you'll return to the same screen. You can complete each of these in turn without repeating the steps above, if you choose.

The **sybinit** program does not begin making changes until you type Control-A from this screen, and then respond "y" to the question "Execute the SQL Server Configuration now?", so typing the incorrect response to any of the prompts or menus does not start the installation.

Steps to Configure Languages

To configure languages, choose:

`6. CONFIGURE LANGUAGES`

The next screen lists the language modules available in your SYBASE directory. You can choose to:

- Remove an existing language from the SQL Server
- Install a language on the server
- Make a language the default on the server

Choose the number of a language from the menu, and reply to each prompt.

Steps to Configure Character Sets

To configure character sets, choose:

```
7. CONFIGURE CHARACTER SETS
```

The next screen lists the character sets available on your system.

You can choose to:

- Remove an already-installed character set
- Install additional character sets
- Make a character set the default for the Server

Steps to Configure a Sort Order

To configure a sort order, choose:

```
8. CONFIGURE SORT ORDER
```

Choose the sort order you wish to install from the menu.

Installing the Changes

Once you have chosen the languages, character sets, and / or sort order for your server, type Control-A to accept the changes. At the prompt:

```
Execute the SQL Server Configuration now?
```

type “y”.

The installation program then boots SQL Server if it is not already running, installs the new values, installs system messages in any language added, and stops and starts SQL Server again to reconfigure. As **sybinit** completes each configuration step, it prints status messages. When the process is finished, this message appears:

```
SQL Server successfully reconfigured.
```

The recovery process following reconfiguration is described in the next section.

Setting Users’ Default Languages

If an additional language is installed, users can run **sp_modifylogin** to set that language as their default language rather than SQL Server’s default language. Installing additional character sets allows clients on platforms that use those character sets to communicate with this SQL Server using character set conversion. See Chapter 18, “Converting Character Sets Between SQL Server and Client” for more information on character set conversion.

Database Dumps and Configuration Changes

➤ ***Note***

Back up all databases on your server before and after you change sort orders or character sets.

If the old sort order and the new sort order are not the same, you are not allowed to load a database from a dump performed before the sort order was reconfigured. If you attempt to do so, an error message appears and the load is aborted. If the character set was reconfigured, and either the old or the new sort order is not binary, you are not allowed to load a database dump made before the character set was reconfigured. However, if both the old and new character sets use binary sort order, you can restore your databases from backups made before the character set was reconfigured.

◆ ***WARNING!***

Do not load a database dump made prior to changing character sets if you have 8-bit character data in that dump. SQL Server interprets all data loaded from a database dump as if it is in the new character set.

If you installed additional languages but did not change SQL Server’s sort order or character set, you have completed the reconfiguration process. If you changed SQL Server’s default sort order or character set, go on to the next section.

If You Changed the Sort Order or Default Character Set

This section describes recovery after reconfiguration and the steps you may need to follow if you changed SQL Server’s sort order or default character set.

If you changed sort orders, you need to do the following after reconfiguring SQL Server:

- Run **sp_indsuspect** to find user indexes that may no longer be valid.
- Rebuild suspect user indexes using the **dbcc reindex** command.

For more information, see ‘‘Using sp_indsuspect to Find Corrupt Indexes’’ on page 17-8, and‘‘Rebuilding Indexes with dbcc reindex’’ on page 17-9.

If you changed to a multibyte character set from any other character set, you must upgrade any existing *text* values with **dbcc fix_text** (when moving to a multibyte character set from either).

See ‘‘Upgrading text Data with dbcc fix_text’’ on page 17-10 for more information.

Recovery after Reconfiguration

Every time SQL Server is stopped and restarted, recovery is performed automatically on each database. Automatic recovery is covered in detail in Chapter 7, ‘‘Developing a Backup and Recovery Plan’’.

After recovery is complete, the new sort order and character set definitions are loaded.

If the sort order has been changed, SQL Server switches to single user mode to allow the necessary updates to system tables and to prevent other users from using the server. Each system table with a character-based index is automatically checked to see if any indexes have been corrupted by the sort order change. Character-based indexes in system tables are automatically rebuilt, if necessary, using the new sort order definition.

After system indexes are rebuilt, character-based user indexes are marked as “suspect” in the *sysindexes* system table without being checked. User tables with suspect indexes are marked as “read only” in *sysobjects* in order to prevent updates to these tables or use of the suspect indexes until the indexes have been checked and rebuilt if necessary.

Next, the new sort order information replaces the old information in the area of the disk that holds configuration information. SQL Server then shuts down to allow it to start with a clean slate for the next session.

Using sp_indsuspect to Find Corrupt Indexes

After SQL Server shuts down, restart it and use **sp_indsuspect** to find the user tables that need to be reindexed. The syntax is:

```
sp_indsuspect [table_name]
```

table_name is the optional name of a specific table. If *table_name* is missing, then **sp_indsuspect** creates a list of all tables in the current database that have indexes marked suspect when the sort order changed. In this example, running **sp_indsuspect** in *mydb* database yields one suspect index:

```
sp_indsuspect

Suspect indexes in database mydb
Own.Tab.Ind (Obj_ID, Ind_ID) =
dbo.holdings.h_name_ix(160048003, 2)
```

Rebuilding Indexes with dbcc reindex

dbcc reindex allows the System Administrator or table owner to check the integrity of indexes attached to a user table and to rebuild suspect indexes. **dbcc reindex** checks each index by running a "fast" version of **dbcc checktable**. The function prints a message when it discovers the first index-related error, then rebuilds the inconsistent indexes.

The syntax is:

dbcc reindex(*table_name* | *table_id*)

Run this utility on all tables listed as containing suspect indexes by **sp_indsuspect**. Here's an example:

```
dbcc reindex(titles)

One or more indexes are corrupt. They will be
rebuilt.
```

In the above example, **dbcc reindex** has discovered one or more suspect indexes in the table *titles*. The **dbcc** utility drops and recreates the appropriate indexes.

If the indexes for a table are already correct, or if there are no indexes for the table, **dbcc reindex** does not rebuild any indexes. It prints an informational message instead. The command is also aborted if a table is suspected of containing corrupt data. If that happens, an error message appears instructing the user to run **dbcc checktable**.

When **dbcc reindex** finishes successfully, all "suspect" marks on the table's indexes are removed. The "read only" mark on the table is removed as well, and the table can be updated. These marks are removed whether or not any indexes have to be rebuilt.

dbcc reindex does not perform reindexing of system tables. System indexes are checked and rebuilt, if necessary, as an automatic part of recovery after SQL Server is restarted following a sort order change.

Upgrading text Data with dbcc fix_text

➤ ***Note***

You must run **dbcc fix_text** if you are changing to a new multibyte character set from either a single-byte or a multibyte character set. You need to run **dbcc fix_text** only on tables that contain text data.

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, SQL Server must be able to handle characters that may span across page boundaries. To do so, the server requires additional information on each of the *text* pages.

To see the names of all tables that contain *text* data, use this query:

```
select sysobjects.name
from sysobjects, syscolumns
where syscolumns.type = 35
and sysobjects.id = syscolumns.id
```

The System Administrator or table owner must run **dbcc fix_text** to calculate the new values needed.

The syntax of **dbcc fix_text** is:

dbcc fix_text (*table_name* | *table_id*)

The table named must be in the current database. **dbcc fix_text** opens the specified table and for each *text* value, calculates the character statistics required and adds them to the appropriate page header fields. This process can take a long time, depending on the number and size of *text* values in a table. **dbcc fix_text** can generate a large number of log records, and may fill up the transaction log. **dbcc fix_text** performs update in a series of small transactions, so if a log becomes full, only a small amount of work is lost. If you run out of log space, clear out your log (see Chapter 8, ‘‘Backing Up and Restoring User Databases’’, for details), and restart **dbcc fix_text** using the same table that was being upgraded when the original **dbcc fix_text** halted. Since each multibyte text value contains information that indicates whether or not it has been upgraded, **dbcc fix_text** only upgrades the text values that were not processed in earlier passes.

If your database stores its log on a separate segment, you can use thresholds to manage clearing the log. See Chapter 10, ‘‘Managing Free Space with Thresholds’’.

If **dbcc fix_text** does not complete updating a table, the utility generates an error message, such as:

```
Not all of the TEXT pages in tablename have been
successfully updated, however, dbcc fix_text is
restartable. Please issue the command again once
any other errors have been addressed.
```

If the utility is unable to acquire a needed lock on a text page, it reports the problem and continues with the work, like this:

```
Unable to acquire an exclusive lock on text page
408. This text value has not been recalculated.
In order to recalculate those TEXT pages you must
release the lock and reissue the dbcc fix_text
command.
```

Retrieving text Values After Changing Character Sets

If you attempt to retrieve *text* values after changing to a multibyte character set and 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:

```
SQL Server is now running a multi-byte character
set,and this TEXT column’s character counts have
not been recalculated using this character set.
Use dbcc fix_text before running this query again.
```

Installing Date Strings for Unsupported Languages

You can use **sp_addlanguage** to install names for the days of the week and months of the year for languages that do not have language modules. With **sp_addlanguage** you define:

- A language name, and optionally, an alias for the name
- A list of the full names of months, and a list of 3-letter abbreviations for the month names
- A list of the full names of the days of the week
- The date format for entering dates (such as month / day / year)
- The number of the first day of the week

This example adds the information for Spanish:

```
sp_addlanguage spanish, espanol,
"enero,febrero,marzo,abril,mayo,junio,julio,agusto,septiembre,oct
ubre,noviembre,diciembre",
"ene,feb,mar,abr,mayo,jun,jul,agu,sep,oct,nov,dic",
"lunes,martes,miercoles,jueves,viernes,sabado,domingo",
dmy, 1
```

sp_addlanguage enforces strict data entry rules. The lists of month names, month abbreviations and days of the week must be comma-separated lists with no spaces or line feeds (returns), and they must contain exactly the right number of elements (12 for the month strings, 7 for the day-of-the-week strings.)

Valid values for the date formats are: *mdy*, *dmy*, *ymd*, *ydm*, *myd*, or *dym*. "dmy" indicates that dates are in day / month / year order. This format affects only data entry; to change output format, you must use the **convert** function.

Server versus Client Side Date Interpretation

When a user selects date values, SQL Server sends them to the client in internal format, and the client converts them for display, using the *common.loc* file and other localization files in the corresponding language subdirectory of the *locales* directory. When a user's default language is set to a language that has been added with **sp_addlanguage**, the client tries to find the correct *common.loc* file for that language. If the client fails to find the correct file, it usually prints an error message, and still connects to the server. If a client connects without having found the right localization files, and then the user selects date values, the dates are displayed using the date strings for the server's default language. You can use the **convert** function to force the dates to be converted to character data at the server.

For example, if a user whose default language is "spanish" logged in from a client that couldn't find Spanish localization files, this query would display the dates in SQL Server's default language:

```
select pubdate from titles
```

The following query returns the date using the Spanish month names:

```
select convert(char(19),pubdate) from titles
```

18 Converting Character Sets Between SQL Server and Client

Introduction

Clients that use different character encoding schemes can connect over a network to SQL Server. In a Western European setting, for example, a server that runs in an ISO 8859-1 (iso_1) environment may be connected to a client that runs in a Roman 8 (roman8) environment. In Japan, a server that runs in an EUC JIS (eucjis) environment may be connected to a client that runs in a Shift-JIS (sjis) environment. This chapter describes the character set conversion features of SQL Server and of the utilities **isql**, **bcp**, and **defncopy**.

Conversion Paths Supported

SQL Server supports conversion among a variety of character sets, as shown in *Table 18-1: Character Set Conversions*. Conversion among the European character sets ISO 8859-1 (iso_1), Code Page 850 (cp850), Code Page 437 (cp437), Roman 8 (roman8), and Macintosh (mac) is supported. Conversion among the Japanese character sets EUC JIS (eucjis), Shift-JIS (sjis), and DEC Kanji (deckanji) is also supported.

An additional character set, ascii_7, is compatible with all character sets. If either the SQL Server or client's character set is ascii_7, any 7-bit ASCII character is allowed to pass between client and server unaltered. Other characters produce conversion errors.

Client's Character Set (columns) / **SQL Server's Character Set** (rows)

	ascii_8	ascii_7	iso_1	cp850	cp437	roman8	mac	eucjis	sjis	deckanji
ascii_8	n / a	7-bit okay								
ascii_7	7-bit okay	n / a	7-bit okay	7-bit okay	7-bit okay	7-bit okay	7-bit okay	7-bit okay	7-bit okay	7-bit okay
iso_1		7-bit okay	n / a							
cp850		7-bit okay		n / a						
cp437		7-bit okay			n / a					
roman8		7-bit okay				n / a				
mac		7-bit okay					n / a			
eucjis		7-bit okay						n / a		
sjis		7-bit okay							n / a	
deckanji		7-bit okay								n / a

(light grey)	No conversions required because SQL Server and client are using the same character set.
(white)	This conversion is supported. Certain types of characters cannot be converted.
(dark grey)	Conversion is not supported between these two character sets.
7-bit okay	Only 7-bit ASCII characters are passed through; others cause a conversion error. The language for the session is also forced to us_english.

Table 18-1: Character Set Conversions

Characters that Cannot Be Converted

In converting any character set to another, there are some characters that cannot be converted. Here are two possibilities:

- The character exists (is encoded) in the source character set, but it does not exist in the target character set. For example, the character "Œ," the OE ligature, is part of the Macintosh character set (codepoint 0xCE). This character does not exist in the ISO 8859-1 character set. If "Œ" exists in data being converted from the Macintosh to the ISO 8859-1 character set, it causes a conversion error.
- The character exists in both the source and the target character set, but in the target character set the character is represented by a different number of bytes than in the source character set. Figure 18-1 compares the EUC JIS and Shift-JIS encodings for the same sequence of characters in a Japanese environment. Kanji, Hiragana, Hankaku Romaji, Zenkaku Romaji, and Zenkaku Katakana characters are represented by the same number of bytes in both character sets and can be converted between EUC JIS and Shift-JIS. However, Hankaku Katakana characters (the last set of characters in the example) are represented by two bytes in EUC JIS and by a single byte in Shift-JIS: these characters cannot be converted.

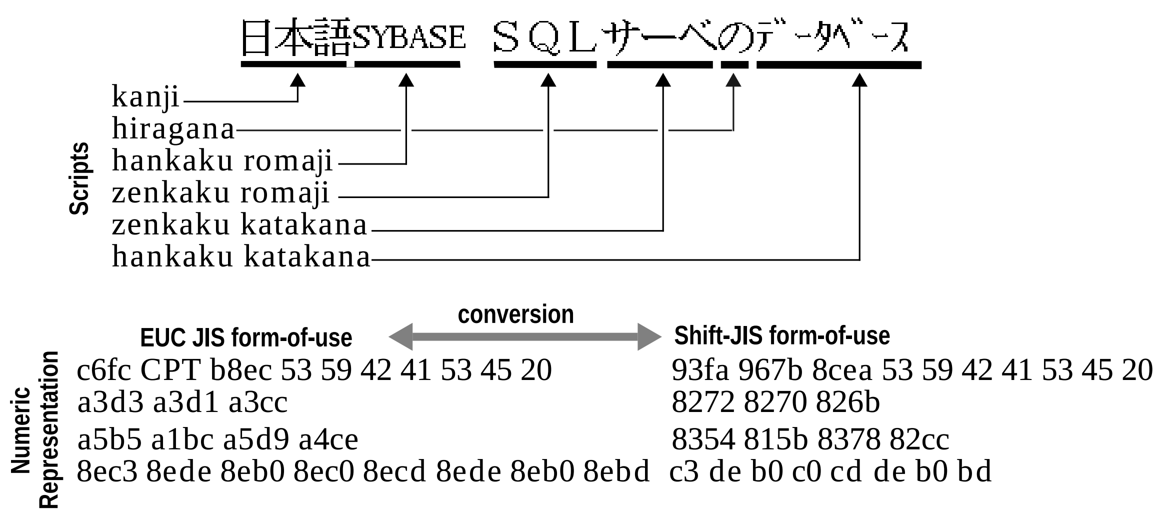


Figure 18-1: Comparison of EUC JIS and Shift-JIS Encoding for Japanese Characters

In addition to Hankaku Katakana, user-defined characters (Gaiji) cannot be converted between Japanese character sets.

Error Handling in Character Set Conversion

SQL Server's character set conversion filters report conversion errors when a character exists in the client's character set but not in the server's, or vice versa. SQL Server must guarantee that data successfully converted on input to the server can be successfully converted back to the client's character set when the client retrieves that data. To do this effectively, SQL Server must avoid putting suspect data into the database.

On encountering a conversion error in data being entered, SQL Server generates this error message:

```
Msg 2402, Severity 16 (EX_USER):
Error converting client characters into server's
character set. Some character(s) could not be
converted.
```

A conversion error prevents query execution.

When SQL Server encounters a conversion error while sending data to the client, the bytes of the suspect characters are replaced by ASCII question marks ("?"). However, the query batch continues to completion. When the statement is complete, SQL Server sends the following message:

```
Msg 2403, Severity 16 (EX_INFO):
WARNING! Some character(s) could not be converted
into client's character set. Unconverted bytes
were changed to question marks ('?').
```

See "Controlling Character Conversion During a Session" on page 18-6 to learn how to turn off error reporting for data being sent from server to client.

Setting up the Conversion Process

Character set conversion begins at login or when the client requests conversion with the **set char_convert** command during a work session. If the client is using a 4.6 or later release of Open Client DB-Library, and the client and SQL Server are using different character sets, conversion is turned on during the login process and set to a default based on the character set that the client is using. Character set conversion can be controlled in the standalone utilities **isql**, **bcp**, and **defncopy** with a command line option. See "Display and File Character Set Command Line Options" on page 18-7 for details.

When a client requests a connection, SQL Server checks whether it can convert from the client's character set to its own character set. If it can, it sets up the appropriate character set conversion filters so that any character data read from or sent to the client automatically passes through them. Next, SQL Server checks the user name and password. These have already been read and must be converted. If they cannot be converted, the login is denied. If conversion on the name and password succeeds, SQL Server looks up the converted strings in *syslogins*.

If SQL Server cannot perform the requested conversions, it sends an error message to the client. The initial message explains why conversion cannot be done and is followed by this message:

```
Msg #2411, Severity 10 (EX_INFO):
No conversions will be done.
```

SQL Server next tries to find the user name and password in their unconverted form in *syslogins*. If it cannot find them, the login is denied. If it succeeds, the login is granted but no character set conversion takes place.

➤ ***Note***

Machine names, user names, and passwords in heterogeneous environments should be composed entirely of 7-bit ASCII characters. If the client's request for character set conversion fails, the login still succeeds if SQL Server finds the unconverted user name and password in *syslogins*.

If the request for conversion fails, or if the client's character set is set to ascii_7, the language for the session is forced to us_english. If the user had requested a different language, an informational message appears, stating that the language for the session is being forced to us_english.

Specifying the Character Set for Utility Programs

A command line option for the **isql**, **bcp**, and **defncopy** utilities specifies the client's character set.

Here are the choices:

- **-J** *charset_name (*UNIX and PC*)* or **/clientcharset** = *charset_name* (OpenVMS) sets the client's character set to the *charset_name*.

- **-J** or **/clientcharset** with no character set name sets the client's character set to NULL. No conversion takes place. No message is sent when this happens.

Omitting the client character set command line flag sets the character set to a default for the platform. This default may not be the character set that the client is using. See the *SQL Server Utility Programs* for information on the default for your platform.

➤ ***Note***

The **-J** flag is used differently on pre-4.9 SQL Servers running on OS/2.

Controlling Character Conversion During a Session

set char_convert allows the user to decide how character set conversion operates during a particular work session. Use **set char_convert** to:

- Set character set conversion on or off
- Start conversion into a specific character set
- Turn error reporting on or off

The syntax for **set char_convert** is:

```
set char_convert {off |
                 {on [with {error | no_error}]} |
                 charset [with {error | no_error}]}
```

Depending on the arguments, the command:

- Turns character set conversion on and off between SQL Server and a client, when used with **off** or **on**:

  ```
  set char_convert on
  ```

  set char_convert off turns conversion off so that characters are sent and received unchanged. **set char_convert on** turns conversion back on after it was turned off. If character set conversion was not turned on during the login process or by the **set char_convert *charset*** command, then **set char_convert on** generates an error message.

- Turns the printing of error messages when the **with no_error** option is included. When a user chooses **with no_error**, SQL Server does not notify the application when characters from SQL Server cannot be converted to the client's character set. Error reporting is initially set to "on" when a client connects with SQL Server: if you do not want error reporting, you must turn it off for each session.

To turn error reporting back on within a session, use the **on with error** option.

Whether or not error reporting is turned on, the bytes which cannot be converted are replaced with ASCII question marks ("?").

- Starts conversion between the server character set and a different client character set, when used with a *charset* value. *charset* can be either the character set's *id* or *name* from *syscharsets*:

  ```
  set char_convert "cp850"
  ```

  If you request character set conversion with **set char_convert** *charset* and SQL Server cannot perform the requested conversions, the session reverts to the state of conversion in effect before the request. For example, if character set conversion is off prior to the **set char_convert** *charset* command, conversion remains off if the request fails.

If the user is using a language other than us_english before entering this command:

```
set char_convert "ascii_7"
```

the language for the session is forced to us_english and an informational message appears. No message appears if the session is already in us_english.

Display and File Character Set Command Line Options

Although the focus of this chapter is on character set conversion between client and SQL Server, character set conversion may be needed in two other places:

- Between the client and a terminal, or
- Between the client and a file system. *Figure 18-2: Where Character Set Conversion May be Needed* illustrates these different paths, as well as the command line options available in the standalone utilities **isql**, **bcp**, and **defncopy**.

As described earlier, the **-J** or **/clientcharset** command line option specifies the character set that the client uses when sending and receiving character data to and from the server.

Setting the Display Character Set

Use the **-a** command line option (**/dispcharset** on OpenVMS) if you are running the client from a terminal with a character set that differs from the client character set. In this case, the **-a** option and the **-J** option are used together to identify the character set translation file (*.xlt* file) needed for the conversion.

Use the **-a** command line option (**/dispcharset** on OpenVMS) without **-J (/clientcharset** on OpenVMS) only if the client character set is the same as the default character set.

Setting the File Character Set

Use the **-q** command line option (**/filecharset** on OpenVMS) if you are running **bcp** to copy character data to or from a file system that uses a character set different from the client character set. In this case, use the **-q** or **/filecharset** option and the **-J** or **/clientcharset** option together to identify the character set translation file (*.xlt* file) needed for the conversion.

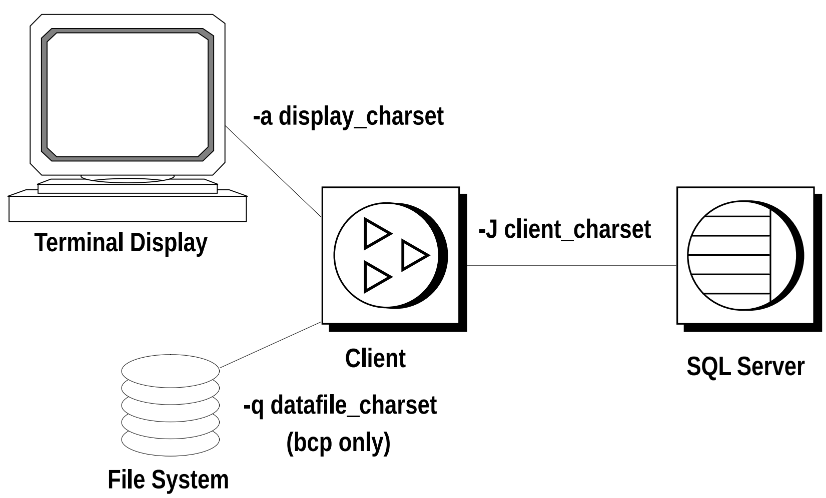


Figure 18-2: Where Character Set Conversion May be Needed

Appendixes

A Reserved Words

Keywords are words that have a special meaning. This appendix lists Transact-SQL, APT-SQL, and SQL92 keywords.

Transact-SQL Reserved Words

The following words are reserved by SQL Server as keywords (command verbs) and cannot be used for the names of database objects such as databases, tables, rules, and defaults. Reserved words can be used for the names of local variables and for stored procedure parameter names.

add	close	disk	from
all	clustered	distinct	goto
alter	commit	double	grant
and	compute	drop	group
any	confirm	dummy	having
arith_overflow	constraint	dump	holdlock
as	continue	else	identity
asc	controlrow	end	identity_insert
at	convert	endtran	if
authorization	count	errlvl	in
avg	create	errorexit	index
begin	current	escape	insert
between	cursor	except	intersect
break	data_pgs	exec	into
browse	database	execute	is
bulk	dbcc	exists	isolation
by	deallocate	exit	key
cascade	declare	fetch	kill
char_convert	default	fillfactor	level
check	delete	for	like

Table A-1: Transact-SQL reserved words

checkpoint	desc	foreign	lineno
load	perm	rollback	to
max	permanent	rowcnt	tran
min	plan	rowcount	transaction
mirror	precision	rows	trigger
mirrorexit	prepare	rule	truncate
national	primary	save	tsequal
noholdlock	print	schema	union
nonclustered	privileges	select	unique
not	proc	set	update
null	procedure	setuser	used_pgs
numeric_truncation	processexit	shared	user
of	public	shutdown	user_option
off	raiserror	some	using
offsets	read	statistics	values
on	readtext	stripe	varying
once	reconfigure	sum	view
only	references	syb_identity	waitfor
open	replace	syb_restree	where
option	reserved_pgs	table	while
or	return	temp	with
order	revoke	temporary	work
over	role	textsize	writetext

Table A-1: Transact-SQL reserved words (continued)

APT-SQL Keywords

Table A-2: APT-SQL keywords lists the APT-SQL keywords which are not reserved words in Transact-SQL. If you are planning to use APT-SQL, avoid using these words as identifiers.

$channel	charindex	int	smallint
$curfield	closesql	interruptsql	sqlbegin
$curform	connect	list	sqlend
$curgroup	curindex	local	sqlexpr
$curpick	cursqlindex	log	sqlrow
$date	datalength	lower	submit
$index	datename	mchoice	substring
$status	datepart	menu	switch
abort	datetime	menubar	switchend
and	define	money	system
append	disconnect	nextquery	tab
apt	enter	nomsg	text
backtab	entry	opensql	textport
bell	exitform	parentname	tinyint
binary	exp	perform	trace
bit	false	positionform	transfer
call	fetchsql	post	trim
callextern	field	printform	true
callform	float	rchoice	upper
callreport	foreach	remote	useform
cancelform	form	reset	variable
case	global	schoice	
channel	hidden	scroll	
char	image	shared	

Table A-2: APT-SQL keywords

SQL92 Keywords

SQL Server 10.0 includes entry-level SQL92 features. Full SQL92 implementation includes the words listed in the following tables as command syntax. Since upgrading identifiers can be a complex process, we are providing this list for your convenience. The publication of this information does not commit Sybase to providing all of these SQL92 features in subsequent releases, and in addition subsequent releases may include keywords not included in this list.

Table A-3 lists the SQL92 keywords which are not reserved words in Transact-SQL.

absolute	constraints	false
action	corresponding	first
allocate	cross	float
are	current_date	found
assertion	current_time	full
bit	current_timestamp	get
bit_length	current_user	global
both	date	go
cascaded	day	hour
case	dec	immediate
cast	decimal	indicator
catalog	deferrable	initially
char	deferred	inner
character	describe	input
char_length	descriptor	insensitive
character_length	diagnostics	int
coalesce	disconnect	integer
collate	domain	interval
collation	end-exec	join
column	exception	language
connect	external	last
connection	extract	leading

Table A-3: SQL92 keywords that are not Transact-SQL reserved words

left	preserve	time
local	prior	timestamp
lower	real	timezone_hour
match	relative	timezone_minute
minute	restrict	trailing
module	right	translate
month	scroll	translation
names	second	trim
natural	section	true
nchar	session	unknown
next	session_user	upper
no	size	usage
nullif	smallint	value
numeric	space	varchar
octet_length	sql	when
outer	sqlcode	whenever
output	sqlerror	write
overlaps	sqlstate	year
pad	substring	zone
partial	system_user	
position	then	

Table A-3: SQL92 keywords that are not Transact-SQL reserved words (continued)

Potential SQL92 Reserved Words

If you are using the ISO / IEC 9075:1989 standard, also avoid using the words in *Table A-4: Potential reserved words*, as these words may become SQL92 reserved words in the future.

after	modify	routine
alias	new	row
async	none	savepoint
before	object	search
boolean	oid	sensitive
breadth	old	sequence
completion	operation	signal
call	operators	similar
cycle	others	sqlexception
data	parameters	structure
depth	pendant	test
dictionary	preorder	there
each	private	type
elseif	protected	under
equals	recursive	variable
general	ref	virtual
ignore	referencing	visible
leave	resignal	wait
less	return	without
limit	returns	
loop	role	

Table A-4: Potential reserved words

B The System Tables

Introduction

All of the tables in the *master* database are system tables. Some of these tables also occur in user databases—they are automatically created when the **create database** command is issued.

These system tables occur in all databases:

System Table	Contents
sysalternates	One row for each SQL Server user mapped to a database user
syscolumns	One row for each column in a table or view, and for each parameter in a procedure
syscomments	One or more rows for each view, rule, default, trigger, and procedure, giving SQL definition statement
sysconstraints	One row for each referential and check constraint associated with a table or column
sysdepends	One row for each procedure, view, or table that is referenced by a procedure, view, or trigger
sysindexes	One row for each clustered or nonclustered index, and one row for each table with no indexes, and an additional row for each table containing *text* or *image* data.
syskeys	One row for each primary, foreign, or common key; set by user (not maintained by SQL Server)
syslogs	Transaction log
sysobjects	One row for each table, view, procedure, rule, trigger default, log, and (in *tempdb* only) temporary object
sysprocedures	One row for each view, rule, default, trigger, and procedure, giving internal definition
sysprotects	User permissions information
sysreferences	One row for each referential integrity constraint declared on a table or column
sysroles	Maps server-wide roles to local database groups
syssegments	One row for each segment (named collection of disk pieces)

Table B-1: System tables that occur in all databases

System Table	Contents
systhresholds	One row for each threshold defined for the database
systypes	One row for each system-supplied and user-defined datatype
sysusermessages	One row for each user-defined message
sysusers	One row for each user allowed in the database

Table B-1: System tables that occur in all databases (continued)

These system tables occur in the *master* database only:

System Table	Contents
syscharsets	One row for each character set or sort order
sysconfigures	One row for each user-settable configuration parameter
syscurconfigs	Information about configuration parameters currently being used by SQL Server
sysdatabases	One row for each database on SQL Server
sysdevices	One row for each tape dump device, disk dump device, disk for databases, and disk partition for databases
sysengines	One row for each SQL Server engine currently on line
syslanguages	One row for each language (except U.S. English) known to the server
syslocks	Information about active locks
sysloginroles	One row for each server login that possesses a system-defined role
syslogins	One row for each valid SQL Server user account
sysmessages	One row for each system error or warning
sysprocesses	Information about server processes
sysremotelogins	One row for each remote user
syssrvroles	One row for each server-wide role
sysservers	One row for each remote SQL Server
sysusages	One row for each disk piece allocated to a database

Table B-2: System tables that occur in the master database only

These system tables occur in the *sybsecurity* database only:

System Table	Contents
sysaudits	One row for each audit record
sysauditoptions	One row for each global audit option

Table B-3: System tables that occur in the sybsecurity *database only*

In the pages that follow, each system table is described in more detail, including a list of their columns and datatypes. In addition, the indexes and the system procedures that reference a particular table are listed.

The word “reserved” in the column description means that the column is currently not being used by SQL Server.

Permissions for use of the system tables can be controlled by the database owner, just like permissions on any other tables.

The SYBASE installation program sets up permissions so that all users can read the system tables, with the exception of a few fields. (See the *SQL Server Installation Guide* for details.)

All direct updates on system tables are by default not allowed even for the database owner. Instead, SQL Server supplies system procedures to make any normally needed updates and additions to system tables.

You can allow direct updates to the system tables if it becomes necessary to modify them in a way that cannot be accomplished with a system procedure. To accomplish this, a System Security Officer must reset the configuration variable called **allow updates** with the system procedure **sp_configure**, and a System Administrator must then execute the **reconfigure** command. For information, see the *System Administration Guide*.

There are entries in some of the *master* database tables that should not be altered by any user under any circumstances. For example, do not attempt to modify *syslogs* with a **delete**, **update**, or **insert** command. In addition, an attempt to **delete** all rows from *syslogs* will put SQL Server into an infinite loop that eventually fills up the entire database.

Note that aggregate functions cannot be used on virtual tables such as *syslocks* and *sysprocesses*.

A diagram of the system tables and their relationships is included at the back of the bound *System Administration Guide* and the *SQL ServerReference Manuals* Volumes 1 and 2. The diagram is not included with camera-ready copy or CD-ROM versions of the manuals.

sysalternates

(all databases)

Description

sysalternates contains one row for each SQL Server user mapped (or aliased) to a user of the current database. When a user tries to access a database, SQL Server looks for a valid *uid* entry in *sysusers*. If none is found, it looks in *sysalternates.suid*. If the user's *suid* is found there, he or she is treated as the database user whose *suid* is listed in *sysalternates.altsuid*.

On the SQL Server distribution tape, there are no entries in *sysalternates*.

Column	Datatype	Description
suid	*smallint*	Server user ID of user being mapped
altsuid	*smallint*	Server user ID of user to whom another user is mapped

Table B-4: Columns in the sysalternates *table*

Indexes

Unique clustered index on *suid*

Referenced by System Procedures

sp_addalias, sp_adduser, sp_changedbowner, sp_dropalias, sp_dropuser, sp_helpuser

sysauditoptions

(*sybsecurity* database)

Description

sysauditoptions contains one row for each global audit option (options set via **sp_auditoption**).These are the system-wide options only, and do not include database, object, stored procedure, trigger, and user audit options. The default value for each option is 0 or "off." *sysauditoptions* can be accessed only by System Security Officers.

Column	Datatype	Description
optn	*smallint*	Option number. See Table B-6.
value	*smallint*	Current value; one of the following: **off** = 0 **ok** = 1 **fail** = 2 **both** = 3 (where applicable) For error auditing (*optn*=13), the values are: **off** = 0 **nonfatal** = 1 **fatal** = 2 **both** = 3
min	*smallint*	Minimum valid value for this option
max	*smallint*	Maximum valid value for this option
name	*varchar(30)*	Name of option
svalue	*varchar(30)*	String equivalent of the current value: for example, "on", "off", "nonfatal"
comment	*varchar(255)*	Description of option

Table B-5: Columns in the sysauditoptions *table*

Possible values for *optn* are:

Option Number	Description
1	Enable or disable auditing
2	Unused
3	Login auditing

Table B-6: Audit option values and descriptions

Option Number	Description
4	Logout auditing
5	Server boot auditing
6	RPC connection auditing
7	Auditing use of the **set** command to turn roles on and off
8	Auditing commands requiring **sa_role** role
9	Auditing commands requiring **sso_role** role
10	Auditing commands requiring **oper_role** role
12	Auditing commands requiring **navigator** role
13	Error auditing
14	Ad hoc auditing
15	Auditing commands requiring **replication** role

Table B-6: Audit option values and descriptions (continued)

Indexes

None

Referenced by System Procedures

sp_auditoption

sysaudits

(*sybsecurity* database)

Description

The *sysaudits* table contains one row for each audit record.

Column	Datatype	Description
event	*smallint*	Type of event being audited. See Table B-8.
eventmod	*smallint*	Further information about the event. Possible values are: 0 = no modifier for this event 1 = successful occurrence of this event; for error auditing (*event*=13), a nonfatal error 2 = failed occurrence of this event; for error auditing (*event*=13), a fatal error
spid	*smallint*	Server process ID of the process that caused the audit record to be written
eventtime	*datetime*	Date and time of the audited event
sequence	*smallint*	Sequence number of the record within a single event; some events require more than one audit record
suid	*smallint*	Server login ID of the user who performed the audited event
dbid	*int null*	Database ID in which the audited event occurred or the object / stored procedure / trigger resides, depending on the type of event
objid	*int null*	ID of the accessed object or stored procedure / trigger
xactid	*binary(6) null*	ID of the transaction containing the audited event. For a multi-database transaction, this is the transaction ID from the database where the transaction originated.
loginname	*varchar(30) null*	Login name corresponding to the *suid*
dbname	*varchar(30) null*	Database name corresponding to the *dbid*
objname	*varchar(30) null*	Object name corresponding to the *objid*

Table B-7: Columns in the sysaudits *table*

Column	Datatype	Description
objowner	*varchar(30) null*	Name of the owner of *objid*
extrainfo	*varchar(255) null*	Additional information about the audited event; contents vary with the type of event audited. (See Table B-8.)

Table B-7: Columns in the sysaudits *table*

Possible values for the *event* column are shown in Table B-8, along with the corresponding contents of the *extrainfo* column. Global audit events have a code of less than 100. All other event types are numbered starting at 100.

event No.	Description	Contents of *extrainfo* Column
1	Enable auditing	NULL
2	Disable auditing	NULL
3	Login	Host name
4	Logout	Host name
5	Server boot	Names of the server program, master device, interfaces file path, server, and error log file
6	RPC connection	Remote server name, host name
7	Use of **set** command to turn roles on and off	Role, new setting
8	Command requiring **sa_role** role	Command type
9	Command requiring **sso_role** role	Command type
10	Command requiring **oper_role** role	Command type
12	Command requiring **navigator** role	Command type
13	Error	Error number, severity, and state
14	Ad hoc audit record	User-supplied comment text
15	Command requiring **replication** role	Command type
100	Database reference	Command type
101	Table reference	Command type
102	View reference	Command type
103	Stored procedure execution	Parameter list
104	Trigger execution	NULL
105	User’s attempts to access a table	Command type
106	User’s attempt to access a view	Command type
107	User’s command text auditing	Command batch text

Table B-8: Contents of event *and* extrainfo *columns of* sysaudits

Indexes

None

Referenced by System Procedures

None

syscharsets

(*master* database only)

Description

syscharsets contains one row for each character set and sort order defined for use by SQL Server. One of the sort orders is marked in *master..sysconfigures* as the default sort order, which is the only one actually in use.

Column	Datatype	Description
type	*smallint*	The type of entity this row represents. Numbers from 1001 to 1999 represent character sets. Numbers from 2000 to 2999 represent sort orders.
id	*tinyint*	The ID for a character set or sort order. A sort order is defined by the combination of the sort order ID and the character set ID (*csid*). The character set is defined by id, which must be unique. Sybase reserves ID numbers 0-200.
csid	*tinyint*	If the row represents a character set, this field is unused. If the row represents a sort order, this is the ID of the character set that sort order is built on. A character set row with this ID must exist in this table.
status	*smallint*	Internal system status information bits.
name	*varchar(30)*	A unique name for the character set or sort order. Must contain only the 7-bit ASCII letters A-Z or a-z, digits 0-9, and underscores (_), and begin with a letter.
description	*varchar(255)*	An optional description of the features of the character set or sort order.
definition	*image*	The internal definition of the character set or sort order. The structure of the data in this field depends on the *type*.

Table B-9: Columns in the syscharsets *table*

Indexes

unique clustered index on *id*, *csid*, *type*
unique nonclustered index on *name*

Referenced by System Procedures

sp_checkreswords, sp_helpsort, sp_serverinfo

syscolumns

(all databases)

Description

syscolumns contains one row for every column in every table and view, and a row for each parameter in a procedure.

Column	Datatype	Description
id	*int*	ID of table to which this column belongs or of procedure with which this parameter is associated
number	*smallint*	Sub-procedure number when the procedure is grouped (0 for non-procedure entries)
colid	*tinyint*	Column ID
status	*tinyint*	Indicates unique position for *bit* columns, whether NULL values are legal in this column, and if a check constraint exists for the column
type	*tinyint*	Physical storage type; copied from *systypes*
length	*tinyint*	Physical length of data; copied from *systypes* or supplied by user
offset	*smallint*	Offset into the row where this column appears, if negative, this is a variable-length column
usertype	*smallint*	User type ID; copied from *systypes*
cdefault	*int*	ID of the procedure that generates default value for this column
domain	*int*	Constraint ID of the first rule or check constraint for this column
name	*sysname*	Column name
printfmt	*varchar(255)*	Reserved
prec	*tinyint*	Number of significant digits
scale	*tinyint*	Number of digits to the right of the decimal point

Table B-10: Columns in the syscolumns *table*

Indexes

unique clustered index on *id, number, colid*

Referenced by System Procedures

sp_bindefault, sp_bindrule, sp_changegroup, sp_checknames, sp_checkreswords, sp_column_privileges, sp_columns, sp_commonkey, sp_droptype, sp_dropuser, sp_estspace, sp_foreignkey, sp_help, sp_helpconstraint, sp_helpjoins, sp_helprotect, sp_primarykey, sp_rename, sp_special_columns, sp_sproc_columns, sp_statistics, sp_unbindefault, sp_unbindrule

syscomments

(all databases)

Description

syscomments contains entries for each view, rule, default, trigger, table constraint, and procedure. The *text* field contains the original definition statements. If the *text* is longer than 255 bytes, the entries will span rows. Each object can occupy up to 65025 rows.

Column	Datatype	Description
id	*int*	Object ID to which this text applies
number	*smallint*	Sub-procedure number when the procedure is grouped (0 for non-procedure entries)
colid	*tinyint*	Sequence of 255 rows for the object
texttype	*smallint*	0 for system-supplied comment (for views, rules, defaults, triggers, and procedures). 1 for user-supplied comment (users can add entries that describe an object or column)
language	*smallint*	Reserved
text	*varchar(255)*	Actual text of SQL definition statement
colid2	*tinyint*	Indicates next sequence of rows for the object (see *colid* above); object can have up to 255 sequences of 255 rows each.

Table B-11: Columns in the syscomments *table*

Indexes

unique clustered index on *id, number, colid, texttype*

Referenced by System Procedures

sp_helpconstraint, sp_helptext

sysconfigures

(*master* database only)

Description

sysconfigures and *syscurconfigs* contain one row for each user-settable configuration variable.

Column	Datatype	Description
config	*smallint*	Configuration variable number
value	*int*	The user-modifiable value for the variable (being used by SQL Server only if **reconfigure** has been run)
comment	*varchar(255)*	Explanation of the configuration variable
status	*smallint*	Either 1 (dynamic, meaning variable takes effect when **reconfigure** is issued) or 0 (variable takes effect when SQL Server is restarted

Table B-12: Columns in the sysconfigures *table*

Here are the contents of *sysconfigures:*

config	value	comment	status
101	0	Maximum recovery interval in minutes	1
102	1	Allow updates to system tables	1
103	0	Number of user connections allowed	0
104	0	Size of available physical memory in 2k pages	0
105	0	Number of open databases allowed among all users	0
106	0	Number of locks for all users	0
107	0	Number of open database objects	0
108	0	Percentage of remaining memory used for procedure cache	0
109	0	Default fill factor percentage	0
110	0	Average time slice per process in milliseconds	0
111	0	Default database size in megabytes	0
112	0	Tape retention period in days	0
113	0	Recovery flags	0
115	1	Allow triggers to be invoked within triggers	1
116	0	Number of devices	0

Table B-13: Contents of sysconfigures

config	value	comment	status
117	1	Allow remote access	1
118	0	Number of remote logins	0
119	0	Number of remote sites	0
120	0	Number of remote connections	0
121	0	Number of pre-read packets per remote connection	0
122	1001	Upgrade version	1
123	50	Default sortorder ID	0
124	0	Default language	1
125	3	Language cache	0
126	1	Maximum online engines	0
127	1	Minimum online engines	0
128	0	Engine adjust interval	0
129	200	CPU accounting flush interval	1
130	1000	I/O accounting flush interval	1
131	1	Default character set ID	0
134	0	Stack size	0
135	0	System-wide password expiration interval	1
136	100	Audit queue size	0
137	0	Additional netmem	0
138	0	Default network packet size	0
139	0	Maximum network packet size	0
140	0	Number of extent I/O buffers	0
141	5000	Identity burning set factor	0

Table B-13: Contents of sysconfigures *(continued)*

Indexes

unique clustered index on *config*

Referenced by System Procedures

sp_configure

sysconstraints

(all databases)

Description

The *sysconstraints* table has one row for each referential and check constraint associated with a table or column.

Whenever a user declares a new check constraint or referential constraint using **create table** or **alter table**, SQL Server inserts a row into the *sysconstraints* table. The row remains until a user executes **alter table** to drop the constraint. Dropping a table by executing **drop table** removes all rows associated with that table from the *sysconstraints* table.

Column	Datatype	Description
colid	*tinyint*	Column number in the table
spare1	*tinyint*	Unused
constrid	*int*	Object ID of the constraint
tableid	*int*	ID of the table on which the constraint is declared
error	*int*	Constraint specific error message
status	*int*	The type of constraint: 0x0040 = a referential constraint 0x0080 = a check constraint
spare2	*int*	Unused

Table B-14: Columns in the sysconstraints *table*

Indexes

clustered index on *tableid, colid*
unique nonclustered index on *constrid*

Referenced by System Procedures

sp_bindmsg, sp_bindrule, sp_helpconstraint, sp_unbindmsg, sp_unbindrule

syscurconfigs

(*master* database only)

Description

syscurconfigs is built dynamically when queried. Its structure is identical to that of *sysconfigures*. It contains an entry for each of the configuration variables, as does *sysconfigures*, but with the current values rather than the default values. In addition, it contains four rows that describe the configuration structure.

Column	Datatype	Description
config	*smallint*	Configuration variable number
value	*int*	The user-modifiable value for the variable (being used by SQL Server only if **reconfigure** has been run)
comment	*varchar(255)*	Explanation of the configuration variable
status	*smallint*	Either 1 (dynamic, meaning variable takes effect when **reconfigure** is issued) or 0 (variable takes effect when SQL Server is restarted

Table B-15: Columns in the syscurconfigs *table*

Referenced by System Procedures

sp_configure, sp_helpsort, sp_serverinfo

sysdatabases

(*master* database only)

Description

sysdatabases contains one row for each database on SQL Server. When SQL Server is installed, *sysdatabases* contains entries for the *master* database, the *model* database, the *sybsystemprocs* database and the *temporary* database. If you have installed auditing, it also contains an entry for the *sybsecurity* database

Column	Datatype	Description
name	*sysname*	Name of the database
dbid	*smallint*	Database ID
suid	*smallint*	Server user ID of database creator
status	*smallint*	Control bits; those which the user can set with **sp_dboption** are marked "settable." See Table B-17.
version	*smallint*	Version of SQL Server code under which database was created
logptr	*int*	Pointer to transaction log
crdate	*datetime*	Creation date
dumptrdate	*datetime*	Date of the last **dump transaction**
status2	*intn*	Additional control bits. See Table B-18.
audflags	*intn*	Audit settings for database
deftabaud	*intn*	Bit-mask that defines default audit settings for tables
defvwaud	*intn*	Bit-mask that defines default audit settings for views
defpraud	*intn*	Bit-mask that defines default audit settings for stored procedures

Table B-16: Columns in the sysdatabases *table*

The bit representations for the *status* column are:

Decimal	Hex	Status
4	0x04	**select into/bulkcopy**; settable
8	0x08	**trunc log on chkpt**; settable
16	0x10	**no chkpt on recovery**; settable
32	0x20	crashed while loading database, instructs recovery not to proceed
256	0x100	database suspect; not recovered; cannot be opened or used; can only be dropped with **dbcc dbrepair**
512	0x200	**ddl in tran**; settable
1024	0x400	**read only**; settable
2048	0x800	**dbo use only**; settable
4096	0x1000	**single user**; settable
8192	0x2000	**allow nulls by default**; settable
16384	0x4000	*dbname* has changed

Table B-17: status *control bits in the* sysdatabases *table*

The bit representations for the *status2* column are:

Decimal	Hex	Status
1	0x01	**abort tran on log full**; settable
2	0x02	**no free space acctg**; settable
4	0x04	**auto identity**; settable

Table B-18: status2 *control bits in the* sysdatabases *table*

Indexes

unique clustered index on *name*
unique nonclustered index on *dbid*

Referenced by System Procedures

sp_addlogin, sp_addsegment, sp_addtype, sp_changedbowner, sp_checknames, sp_checkreswords, sp_databases, sp_dboption, sp_dbremap, sp_dropdevice, sp_dropsegment, sp_extendsegment, sp_helpdb, sp_logdevice, sp_renamedb, sp_tables

sysdepends

(all databases)

Description

sysdepends contains one row for each procedure, view, or table that is referenced by a procedure, view, or trigger.

Column	Datatype	Description
id	*int*	Object ID
number	*smallint*	Procedure number
depid	*int*	Dependent object ID
depnumber	*smallint*	Dependent procedure number
status	*smallint*	Internal status information
selall	*bit*	On if object is used in **select** * statement
resultobj	*bit*	On if object is being updated
readobj	*bit*	On if object is being read

Table B-19: Columns in the sysdepends *table*

Indexes

unique clustered index on *id, number, depid, depnumber*

Referenced by System Procedures

sp_depends

sysdevices

(*master* database only)

Description

sysdevices contains one row for each tape dump device, disk dump device, disk for databases, and disk partition for databases. On the SQL Server distribution tape, there are four entries in *sysdevices*: one for the master device (for databases), one for a disk dump device, and two for tape dump devices.

Column	Datatype	Description
low	*int*	First virtual page number on database device (not used for dump devices)
high	*int*	Last virtual page number on database device or dump device
status	*smallint*	Bit map indicating type of device, default and mirror status. See Table B-21.
cntrltype	*smallint*	Controller type (0 if database device, 2 if disk dump device or streaming tape, 3–8 if tape dump device)
name	*sysname*	Logical name of dump device or of database device
phyname	*varchar(127)*	Name of physical device
mirrorname	*varchar(127)*	Name of mirror device

Table B-20: Columns in the sysdevices *table*

The bit representations for the *status* column are additive. For example, “3” indicates a physical disk that is also a default.

The status control bits are:

Decimal	Hex	Status
1	0x01	default disk
2	0x02	physical disk
4	0x04	logical disk
8	0x08	skip header

Table B-21: status *control bits in the* sysdevices *table*

Decimal	Hex	Status
16	0x10	dump device
32	0x20	serial writes
64	0x40	device mirrored
128	0x80	reads mirrored
256	0x100	secondary mirror side only
512	0x200	mirror enabled
2048	0x800	used internally

Table B-21: status *control bits in the* sysdevices *table*

Indexes

unique clustered index on *name*

Referenced by System Procedures

sp_addsegment, sp_addumpdevice, sp_checknames, sp_checkreswords, sp_configure, sp_diskdefault, sp_dropdevice, sp_dropsegment, sp_extendsegment, sp_helpdb, sp_helpdevice, sp_helplog, sp_helpsegment, sp_logdevice, sp_volchanged

sysengines

(*master* database only)

Description

sysengines contains one row for each SQL Server engine currently on line.

Column	Datatype	Description
engine	*smallint*	Engine number
osprocid	*int*	Operating system process ID (may be NULL)
osprocname	*char*	Operating system process name (may be NULL)
status	*char*	One of: online, off-line, in create, in destroy, debug
affinitied	*int*	Number of SQL Server processes with affinity to this engine
cur_kpid	*int*	Kernel process ID of process currently running on this engine, if any
last_kpid	*int*	Kernel process ID of process which previously ran on this engine
idle_1	*tinyint*	Reserved
idle_2	*tinyint*	Reserved
idle_3	*tinyint*	Reserved
idle_4	*tinyint*	Reserved
starttime	*datetime*	Date and time engine came on line

Table B-22: Columns in the sysengines *table*

Indexes

none

Referenced by System Procedures

sp_monitor

sysindexes

(all databases)

Description

sysindexes contains one row for each clustered index, one row for each nonclustered index, one row for each table that has no clustered index, and one row for each table that contains *text* or *image* columns.

The column *doampg* is used only if the row describes a table or clustered index.

Column	Datatype	Description
name	*sysname*	Index or table name
id	*int*	ID of table, or ID of table to which index belongs
indid	*smallint*	0 if table, 1 if clustered index, >1 if nonclustered, 255 if text chain
doampg	*int*	Page number for the object allocation map of a table or clustered index
ioampg	*int*	Page number for the allocation map of a nonclustered index
oampgtrips	*int*	Ratio of OAM page to data page residency in cache
status2	*int*	Internal system status information. See Table B-24.
ipgtrips	*int*	Ratio of index page to data page residency in cache
first	*int*	Pointer to first data or leaf page
root	*int*	Pointer to root page if entry is an index; pointer to last page if entry is a table or text chain
distribution	*int*	Pointer to distribution page (if entry is an index)
usagecnt	*smallint*	Reserved
segment	*smallint*	Number of segment in which this object resides

Table B-23: Columns in the sysindexes *table*

Column	Datatype	Description
status	*smallint*	Internal system status information (See Table B-25)
rowpage	*smallint*	Maximum number of rows per page
minlen	*smallint*	Minimum size of a row
maxlen	*smallint*	Maximum size of a row
maxirow	*smallint*	Maximum size of a non-leaf index row
keycnt	*smallint*	Number of keys for a clustered index; number of keys+1 for a nonclustered index
keys1	*varbinary(255)*	Description of key columns (if entry is an index)
keys2	*varbinary(255)*	Description of key columns (if entry is an index)
soid	*tinyint*	Sort order ID that the index was created with. ‘0’ if there is no character data in the keys
csid	*tinyint*	Character set ID that the index was created with. ‘0’ if there is no character data in the keys

Table B-23: Columns in the sysindexes *table*

The bit representations for the *status2* column are:

Decimal	Hex	Status
1	0x1	Index part of referential integrity constraint
2	0x2	Index part of primary key / unique constraint

Table B-24: status2 *control bits in the* sysindexes *table*

The bit representations for the *status* column are:

Decimal	Hex	Status
1	0x1	Abort command or trigger if attempt to insert duplicate key
2	0x2	Unique index
4	0x4	Abort command or trigger if attempt to insert duplicate row
16	0x10	Clustered index
64	0x40	Index allows duplicate rows
32768	0x8000	Suspect index

Table B-25: status *control bits in the* sysindexes *table*

Indexes

unique clustered index on *id, indid*

Referenced by System Procedures

sp_checknames, sp_checkreswords, sp_dropsegment, sp_estspace, sp_help, sp_helpconstraint, sp_helpindex, sp_helplog, sp_helpsegment, sp_indsuspect, sp_pkeys, sp_placeobject, sp_rename, sp_spaceused, sp_special_columns, sp_statistics

syskeys

(all databases)

Description

syskeys contains one row for each primary, foreign, or common key.

Column	Datatype	Description
id	*int*	Object ID
type	*smallint*	Record type
depid	*int null*	Dependent object ID
keycnt	*int null*	The number of non-null keys
size	*int null*	Reserved
key1	*int null*	Column ID
key2	*int null*	Column ID
key3	*int null*	Column ID
key4	*int null*	Column ID
key5	*int null*	Column ID
key6	*int null*	Column ID
key7	*int null*	Column ID
key8	*int null*	Column ID
depkey1	*int null*	Column ID
depkey2	*int null*	Column ID
depkey3	*int null*	Column ID
depkey4	*int null*	Column ID
depkey5	*int null*	Column ID
depkey6	*int null*	Column ID
depkey7	*int null*	Column ID
depkey8	*int null*	Column ID

Table B-26: Columns in the syskeys *table*

Indexes

clustered index on *id*

Referenced by System Procedures

sp_commonkey, sp_dropkey, sp_foreignkey, sp_helpjoins, sp_helpkey, sp_primarykey

syslanguages

(*master* database only)

Description

syslanguages contains one row for each language known to SQL Server. *us_english* is not in *syslanguages,* but is always available to SQL Server.

Column	Datatype	Description
langid	*smallint*	Unique language ID
dateformat	*char(3)*	Date order, for example, “dmy”
datefirst	*tinyint*	First day of the week—1 for Monday, 2 for Tuesday, and so on, up to 7 for Sunday.
upgrade	*int*	SQL Server version of last upgrade for this language
name	*varchar(30)*	Official language name, for example, “french”
alias	*varchar(30)*	Alternate language name, for example, “francais”
months	*varchar(251)*	Comma-separated list of full-length month names, in order from January to December—each name is at most 20 characters long
shortmonths	*varchar(119)*	Comma-separated list of shortened month names, in order from January to December—each name is at most 9 characters long
days	*varchar(216)*	Comma-separated list of day names, in order from Monday to Sunday—each name is at most 30 characters long.

Table B-27: Columns in the syslanguages *table*

Indexes

unique clustered index on *langid*
unique nonclustered index on *name*
unique nonclustered index on *alias*

Referenced by System Procedures

sp_addlanguage, sp_addmessage, sp_checkreswords, sp_configure, sp_droplanguage, sp_dropmessage, sp_getmessage, sp_helplanguage, sp_setlangalias

syslocks

(*master* database only)

Description

syslocks contains information about active locks, but it is not a normal table. Rather, it is built dynamically when queried by a user. No updates to *syslocks* are allowed.

Column	Datatype	Description
id	*int*	Table ID
dbid	*smallint*	Database ID
page	*int*	Page number
type	*smallint*	Type of lock *(*bit values for the *type* column are listed in Table B-29)
spid	*smallint*	ID of process that holds the lock
class	*char(30)*	Name of the cursor this lock is associated with, if any

Table B-28: Columns in the syslocks *table*

The bit representations for the type column are:

Decimal	Hex	Status
1	0x1	Exclusive table lock
2	0x2	Shared table lock
3	0x3	Exclusive intent lock (will do page locking on indicated pages)
4	0x4	Shared intent lock
5	0x5	Exclusive page lock
6	0x6	Shared page lock
7	0x7	Update page lock (changes to exclusive if page is actually modified)
256	0x100	Lock is blocking another process
512	0x200	Demand lock

Table B-29: type *control bit in the* syslocks *table*

Indexes

none

Referenced by System Procedures

sp_lock

sysloginroles

(*master* database only)

Description

sysloginroles contains a row for each instance of a server login possessing a system-defined role. One row is added for each role possessed by each login. For example, if a single server user is granted three roles, three rows are added to *sysloginroles* associated with that user's *suid*.

Column	Datatype	Description
suid	*suid*	Server user ID
srid	*smallint*	Server role ID; one of the following: 0 **sa_role** 1 **sso_role** 2 **oper_role** 4 **navigator_role** 5 **replication_role** 6 **bcpin_labels_role**
status	*smallint*	Reserved

Table B-30: Columns in the sysloginroles *table*

Indexes

Clustered index on *suid*

Referenced by System Procedures

sp_displaylogin, sp_droplogin, sp _locklogin, sp_role

syslogins

(*master* database only)

Description

syslogins contains one row for each valid SQL Server user account. On the SQL Server distribution tape, *syslogins* contains an entry in which the name is “sa”, the suid is 1, and the password is null. It also contains an entry named “probe” with an unpublished password. The login “probe” and the user “probe” exist for the Two Phase Commit Probe Process, which uses a challenge and response mechanism to access SQL Server.

Column	Datatype	Description
suid	*smallint*	Server user ID
status	*smallint*	Status of the account. See Table B-32.
accdate	*datetime*	Date *totcpu* and *totio* were last cleared
totcpu	*int*	CPU time accumulated by login
totio	*int*	I/O accumulated by login
spacelimit	*int*	Reserved
timelimit	*int*	Reserved
resultlimit	*int*	Reserved
dbname	*sysname*	Name of database in which to put user when connection established
name	*sysname*	Login name of user
password	*varbinary*	Password of user (encrypted)
language	*varchar(30)*	User’s default language
pwdate	*datetime*	Date the password was last changed
audflags	*int*	User’s audit settings
fullname	*varchar(30)*	Full name of the user

Table B-31: Columns in the syslogins *table*

The bit representations for the *status* column are:

Decimal	Hex	Status
1	0x1	Password less than 6 characters, or NULL
2	0x2	Account is locked
4	0x4	Password is expired

Table B-32: status *control bits in the* syslogins *table*

Indexes

unique clustered index on *suid*
unique nonclustered index on *name*

Referenced by System Procedures

sp_addalias, sp_addlogin, sp_addremotelogin, sp_adduser, sp_changedbowner, sp_checknames, sp_checkreswords, sp_clearstats, sp_displaylogin, sp_droplogin, sp_helpdb, sp_helpuser, sp_locklogin, sp_modifylogin, sp_reportstats, sp_role

syslogs

(all databases)

Description

syslogs contains the transaction log. It is used by SQL Server for recovery and roll forward, and is not useful to users.

You cannot delete from, insert into, or update *syslogs*. Every data modification operation is logged, so before you can change *syslogs*, the change must be logged. This means that a change operation on *syslogs* adds a row to *syslogs*, which then must be logged, adding another row to *syslogs*, and so on, producing an infinite loop. The loop continues until the database becomes full.

Column	Datatype	Description
xactid	*binary(6)*	Transaction ID
op	*tinyint*	Update operation number

Table B-33: Columns in the syslogs *table*

Indexes

none

sysmessages

(*master* database only)

Description

sysmessages contains one row for each system error or warning that can be returned by SQL Server. SQL Server displays the error description on the user's screen.

Column	Datatype	Description
error	*int*	Unique error number
severity	*smallint*	Severity level of error
dlevel	*smallint*	Reserved for number of descriptive level of this message: terse, short, or long
description	*varchar(25)*	Explanation of error with place holders for parameters
langid	*smallint*	Language, null for us_english
sqlstate	*varchar(5)*	SQLSTATE value for the error

Table B-34: Columns in the sysmessages *table*

Indexes

clustered index on *error, dlevel*
unique nonclustered index on *error, dlevel, langid*

Referenced by System Procedures

sp_configure, sp_dboption, sp_depends, sp_droplanguage, sp_getmessage, sp_help, sp_helpdb, sp_helpdevice, sp_helpremotelogin, sp_remoteoption

sysobjects

(all databases)

Description

sysobjects contains one row for each table, view, stored procedure, log, rule, default, trigger, check constraint, referential constraint, and (in *tempdb* only) temporary object.

Column	Datatype	Description
name	*sysname*	Object name
id	*int*	Object ID
uid	*smallint*	User ID of object owner
type	*char(2)*	One of the following object types: S system table U user table V view L log P procedure R rule D default TR trigger RI referential constraint
userstat	*smallint*	Application-dependent type information (32768 decimal [0x8000 hex] indicates to Data Workbench^{TM} that a procedure is a report)
sysstat	*smallint*	Internal status information (256 decimal [0x100 hex] indicates that table is read-only)
indexdel	*smallint*	Index delete count (incremented if an index is deleted)
schemacnt	*smallint*	Count of changes in schema of a given object (incremented if a rule or default is added)
sysstat2	*smallint*	Additional internal status information. See Table B-36.
crdate	*datetime*	Date object was created
expdate	*datetime*	Reserved
deltrig	*int*	Stored procedure ID of a delete trigger
instrig	*int*	Stored procedure ID of an insert trigger

Table B-35: Columns in the sysobjects *table*

Column	Datatype	Description
updtrig	*int*	Stored procedure ID of an update trigger
seltrig	*int*	Reserved
ckfirst	*int*	ID of first check constraint on the table
cache	*smallint*	Reserved
audflags	*int*	Object's audit settings
objspare	*int*	Spare

Table B-35: Columns in the sysobjects *table (continued)*

The bit representations for the *sysstat2* column are:

Decimal	Hex	Status
1	0x1	Table has referential constraint
2	0x2	Table has foreign key constraint
4	0x4	Table has more than one check constraint
8	0x8	Table has primary key constraint
16	0x10	Chained transaction mode only stored procedure
32	0x20	Any transaction mode stored procedure
64	0x40	Table has IDENTITY field

Table B-36: systat2 *control bits in the* sysobjects *table*

Indexes

unique clustered index on *id*
unique nonclustered index on *name, uid*

Referenced by System Procedures

sp_addmessage, sp_addthreshold, sp_bindefault, sp_bindmsg, sp_bindrule, sp_checknames, sp_checkreswords, sp_column_privileges, sp_columns, sp_commonkey, sp_depends, sp_dropgroup, sp_dropkey, sp_dropsegment, sp_dropthreshold, sp_droptype, sp_dropuser, sp_estspace, sp_fkeys, sp_foreignkey, sp_help, sp_helpconstraint, sp_helpindex, sp_helpjoins, sp_helpkey, sp_helprotect, sp_helpthreshold, sp_indsuspect, sp_modifythreshold, sp_pkeys, sp_placeobject, sp_primarykey, sp_procxmode, sp_recompile, sp_remap, sp_rename, sp_spaceused, sp_sproc_columns, sp_statistics, sp_stored_procedures, sp_table_privileges, sp_tables, sp_unbindefault, sp_unbindmsg, sp_unbindrule

sysprocedures

(all databases)

Description

sysprocedures contains entries for each view, default, rule, trigger, procedure, declarative default, and check constraint. The plan or sequence tree for each object is stored in binary form. If the sequence tree doesn't fit in one entry, it is broken into more than one row. The *sequence* column identifies the sub-rows.

Column	Datatype	Description
type	*smallint*	Object type. See Table B-38.
id	*int*	Object ID
sequence	*smallint*	Sequence number if more than one row is used to describe this object
status	*smallint*	Internal system status
number	*smallint*	Sub-procedure number when the procedure is grouped (0 for non-procedure entries)

Table B-37: Columns in the sysprocedures *table*

The bit representations for the *type* column are:

Decimal	Hex	Status
1	0x1	Entry describes a plan (reserved)
2	0x2	Entry describes a tree

Table B-38: type *control bits in the* sysprocedures *table*

Indexes

unique clustered index on *id, type, sequence, number*

Referenced by System Procedures

sp_bindefault, sp_bindrule, sp_remap, sp_sproc_columns, sp_stored_procedures, sp_unbindefault, sp_unbindrule

sysprocesses

(*master* database only)

Description

sysprocesses contains information about SQL Server processes, but it is not a normal table. Rather, it is built dynamically when queried by a user. No updates to *sysprocesses* are allowed.

Use the **kill** statement to kill a process.

Column	Datatype	Description
spid	*smallint*	Process ID
kpid	*int*	Kernel process ID
enginenum	*int*	Number of engine on which process is being executed
status	*char(12)*	Process ID status, one of: infected background recv sleep send sleep alarm sleep lock sleep sleeping runnable running stopped bad status log suspend
suid	*smallint*	Server user ID of user who issued command
hostname	*char(10)*	Name of host computer
program_name	*char(16)*	Name of front-end module
hostprocess	*char(8)*	Host process ID number
cmd	*char(16)*	Command currently being executed
cpu	*int*	Cumulative cpu time for process in ticks
physical_io	*int*	Number of disk reads and writes for current command
memusage	*int*	Amount of memory allocated to process
blocked	*smallint*	Process ID of blocking process, if any

Table B-39: Columns in the sysprocesses *table*

Column	Datatype	Description
dbid	*smallint*	Database ID
uid	*smallint*	ID of user who executed command
gid	*smallint*	Group ID of user who executed command
tran_name	*varchar*(64)	Name of the active transaction
time_blocked	*int*	Time blocked in seconds
network_pktsz	*int*	Current connection's network packet size

Table B-39: Columns in the sysprocesses *table*

Indexes

none

Referenced by System Procedures

sp_dboption, sp_droplogin, sp_locklogin, sp_role, sp_who

sysprotects

(all databases)

Description

sysprotects contains information on user permissions information—entries for each **grant** and **revoke** statement that has been issued.

Column	Datatype	Description
id	*int*	ID of object to which this permission applies
uid	*smallint*	ID of user or group to which this permission applies
action	*tinyint*	One of the following permissions: **select =** 193 **insert** = 195 **delete** = 196 **update** = 197 **execute** = 224 **references** = 151 **create database** = 203 **create default** = 233 **create procedure** = 222 **create rule** = 236 **create table** = 198 **create view** = 207 **dump database** = 228 **dump transaction** = 235
protecttype	*tinyint*	One of the following values: **grant with grant** = 0 **grant** (permanent) = 1 **revoke** (permanent) = 2
columns	*varbinary(32)*	Bit map of columns to which this **select** or update permission applies. Bit 0 indicates all columns; 1 means permission applies to that column; null means no information.
grantor	*smallint*	User ID of the grantor

Table B-40: Columns in the sysprotects *table*

Indexes

unique clustered index on *id, action, grantor, uid, protecttype*

Referenced by System Procedures

sp_changegroup, sp_dropgroup, sp_dropuser, sp_helprotect, sp_stored_procedures, sp_tables

sysreferences

(all databases)

Description

sysreferences contains one row for each referential integrity constraint declared on a table or column.

Column	Datatype	Description
indexid	*smallint*	ID of the unique index on referenced columns
constrid	*int*	Object ID of the constraint from *sysobjects*
tableid	*int*	Object ID of the referencing table
reftabid	*int*	Object ID of the referenced table
keycnt	*smallint*	The number of columns in the foreign key
status	*smallint*	Reserved
frgndbid	*smallint*	Reserved
pmrydbid	*smallint*	Reserved
spare2	*int*	Reserved
fokey1 . . .	*tinyint*	Column ID of the first referencing column
fokey16	*tinyint*	Column ID of the 16th referencing column
refkey1 . . .	*tinyint*	Column ID of the first referenced column
refkey16	*tinyint*	Column ID of the 16th referenced column
frgndbname	*varchar(30)*	Name of the database that includes the referencing table (the table with the foreign key). Null if the referencing table is in the current database.
pmrydbname	*varchar(30)*	Name of the database that includes the referenced table (the table with the primary key). Null if the referenced table is in the current database.

Table B-41: Columns in the sysreferences *table*

Indexes

clustered index on *frgndbname, tableid*
unique nonclustered index on *frgndbname, constrid*
nonclustered index on *pmrydbname, reftabid, indexid*

Referenced by System Procedures

sp_fkeys, sp_helpconstraint

sysremotelogins

(*master* database only)

Description

s*ysremotelogins* contains one row for each remote user who is allowed to execute remote procedure calls on this SQL Server.

Column	Datatype	Description
remoteserverid	*smallint*	Identifies the remote server
remoteusername	*varchar*(30)	User's login name on remote server
suid	*smallint*	Local server user ID
status	*smallint*	Bitmap of options

Table B-42: Columns in the sysremotelogins *table*

Indexes

unique clustered index on *remoteserverid, remoteusername*

Referenced by System Procedures

sp_addremotelogin, sp_checknames, sp_checkreswords, sp_dropremotelogin, sp_dropserver, sp_helpremotelogin, sp_remoteoption

sysroles

(all databases)

Description

sysroles maps server role IDs to local role IDs.

Column	Datatype	Description
id	*smallint*	Server role ID (*srid*)
lrid	*smallint*	Local role ID
type	*smallint*	Unused
status	*smallint*	Unused

Table B-43: Columns in the sysroles *table*

Indexes

Unique clustered index on *lrid*

Referenced by System Procedures

None

syssegments

(all databases)

Description

syssegments contains one row for each segment (named collection of disk pieces). The default entries are: segment 0 (*system*) for system tables; segment 2 *(logsegment*) for the transaction log; and segment 1 (*default*) for other objects. Each database has an entry in *sysusages* contain these segments in its maps.

Column	Datatype	Description
segment	*smallint*	Segment number
name	*sysname*	Segment name
status	*int null*	Indicates which segment is default segment

Table B-44: Columns in the syssegments *table*

Indexes

none

Referenced by System Procedures

sp_addsegment, sp_addthreshold, sp_checknames, sp_checkreswords, sp_dropsegment, sp_dropthreshold, sp_dropuser, sp_extendsegment, sp_helpdb, sp_helpindex, sp_helpsegment, sp_helpthreshold, sp_modifythreshold, sp_placeobject

sysservers

(*master* database only)

Description

sysservers contains one row for each remote SQL Server, Backup Server, or Open Server on which this SQL Server can execute remote procedure calls.

Column	Datatype	Description
srvid	*smallint*	ID number (for local use only) of the remote server
srvstatus	*smallint*	Bitmap of options
srvname	*varchar(30)*	Server name
srvnetname	*varchar(32)*	Interfaces file name for the server

Table B-45: Columns in the sysservers *table*

Indexes

unique clustered index on *srvid*
unique nonclustered index on *srvname*

Referenced by System Procedures

sp_addremotelogin, sp_addserver, sp_checknames, sp_checkreswords, sp_configure, sp_dropremotelogin, sp_dropserver, sp_helpremotelogin, sp_helpserver, sp_remoteoption, sp_serveroption

syssrvroles

(*master* database only)

Description

syssrvroles contains a row for each server-wide role.

Column	Datatype	Description
srid	*smallint*	Server role ID
name	*varchar*	Name of the role

Table B-46: Columns in the syssrvroles *table*

Indexes

Unique clustered index on *srid*

Referenced by System Procedures

sp_adduser, sp_changegroup, sp_displaylogin, sp_dropgroup, sp_helpgroup, sp_role

systhresholds

(all databases)

Description

systhresholds contains one row for each threshold defined for the database.

Column	Datatype	Description
segment	*smallint*	Segment number for which free space is being monitored
free_space	*int*	Size of threshold, in 2K pages (4K for Stratus)
status	*smallint*	Bit 1 equals 1 for the logsegment's last-chance threshold, 0 for all other thresholds
proc_name	*varchar(255)*	Name of the procedure that is executed when the number of unused pages on *segment* falls below *free_space*.
suid	*smallint*	The server user ID of the user who added the threshold or modified it most recently
currauth	*varbinary*(255)	A bit mask that indicates which roles were active for *suid* at the time the threshold was added or most recently modified. When the threshold is crossed, *proc_name* executes with this set of roles, less any that have been deactivated since the threshold was added or last modified.

Table B-47: Columns in the systhresholds *table*

Indexes

Unique clustered index on *segment, free_space*

Referenced by System Procedures

sp_addthreshold, sp_dropsegment, sp_dropthreshold, sp_dropuser, sp_helpthreshold, sp_modifythreshold

systypes

(all databases)

Description

systypes contains one row for each system-supplied and user-defined datatype. Domains (defined by rules) and defaults are given, if they exist.

The rows that describe system-supplied datatypes cannot be altered.

Column	Datatype	Description
uid	*smallint*	User ID of datatype creator
usertype	*smallint*	User type ID
variable	*bit*	1 if datatype is variable length; 0 otherwise
allownulls	*bit*	Indicates whether nulls are allowed for this datatype
type	*tinyint*	Physical storage datatype
length	*tinyint*	Physical length of datatype
tdefault	*int*	ID of system procedure that generates default for this datatype
domain	*int*	ID of system procedure that contains integrity checks for this datatype
name	*sysname*	Datatype name
printfmt	*varchar(255)*	Reserved
prec	*tinyint*	Number of significant digits
scale	*tinyint*	Number of digits to the right of the decimal point
ident	*tinyint*	1 if column has the IDENTITY property, 0 if not
hierarchy	*tinyint*	Precedence of the datatype in mixed mode arithmetic

Table B-48: Columns in the systypes *table*

The listing that follows includes the system-supplied datatype *name*, *hierarchy*, *type* (not necessarily unique), and *usertype* (unique). The datatypes are ordered by *hierarchy*. In mixed mode arithmetic, the datatype with the lowest *hierarchy* takes precedence:

name	hierarchy	type	usertype
floatn	1	109	14
float	2	62	8
datetimn	3	111	15
datetime	4	61	12
real	5	59	23
numericn	6	108	28
numeric	7	63	10
decimaln	8	106	27
decimal	9	55	26
moneyn	10	110	17
money	11	60	11
smallmoney	12	122	21
smalldatetime	13	58	22
intn	14	38	13
int	15	56	7
smallint	16	52	6
tinyint	17	48	5
bit	18	50	16
varchar	19	39	2
sysname	19	39	18
nvarchar	19	39	25
char	20	47	1
nchar	20	47	24
varbinary	21	37	4
timestamp	21	37	80
binary	22	45	3
text	23	35	19
image	24	34	20

Table B-49: Datatype names, hierarchy, types, and usertypes

Indexes

unique clustered index on *name*
unique nonclustered index on *usertype*

Referenced by System Procedures

sp_addtype, sp_bindefault, sp_bindrule, sp_checknames, sp_checkreswords, sp_columns, sp_datatype_info, sp_droptype, sp_dropuser, sp_help, sp_rename, sp_special_columns, sp_sproc_columns,sp_unbindefault, sp_unbindrule

sysusages

(*master* database only)

Description

sysusages contains one row for each disk allocation piece assigned to a database. Each database contains a specified number of database (logical) page numbers. Each disk piece includes the segments on the SQL Server distribution tape, segments 0 and 1.

The **create database** command checks *sysdevices* and *sysusages* to find available disk allocation pieces. One or more contiguous disk allocation pieces is assigned to the database, and the mapping is recorded in *sysusages*.

Column	Datatype	Description
dbid	*smallint*	Database ID
segmap	*int*	Bit map of possible segment assignments
lstart	*int*	First database (logical) page number
size	*int*	Number of contiguous database (logical) pages
vstart	*int*	Starting virtual page number
pad	*smallint*	Unused
unreservedpgs	*int*	Free space not part of an allocated extent

Table B-50: Columns in the sysusages *table*

Indexes

unique clustered index on *dbid, lstar*t
unique nonclustered index on *vstart*

Referenced by System Procedures

sp_addsegment, sp_addthreshold, sp_databases, sp_dropdevice, sp_dropsegment, sp_extendsegment, sp_helpdb, sp_helplog, sp_helpsegment, sp_logdevice, sp_modifythreshold, sp_spaceused

sysusermessages

(all databases)

Description

sysusermessages contains one row for each user-defined message that can be returned by SQL Server.

Column	Datatype	Description
error	*int*	Unique error number. Must be 20000 or above.
uid	*smallint*	User ID of the message creator
description	*varchar(255)*	User-defined message with optional place holders for parameters
langid	*smallint*	Language ID for this message; null for us_english

Table B-51: Columns in the sysusermessages *table*

Indexes

clustered index on *error*
unique nonclustered index on *error, langid*

Referenced by System Procedures

sp_addmessage, sp_bindmsg, sp_dropmessage, sp_getmessage, sp_helpconstraint

sysusers

(all databases)

Description

sysusers contains one row for each user allowed in the database, and one row for each group or role.

On the SQL Server distribution tape, *master..sysusers* contains some initial users: "dbo," whose *suid* is 1 and *uid* is 1; "guest," whose *suid* is -1 and *uid* is 2; and "public," whose *suid* is -2 and *uid* is 0. In addition, each role (**sa_role**, **sso_role**, and so on) is listed in *sysusers*, because SQL Server treats roles much like groups.

The user *guest* provides a mechanism for giving users not explicitly listed in *sysusers* access to the database with a restricted set of permissions. The "guest" entry in *master* means that any user with an account on SQL Server (that is, with an entry in *syslogins*) can access *master*.

The user "public" refers to all users. The keyword **public** is used with the **grant** and **revoke** commands to signify that permission is being given to or taken away from all users.

Column	Datatype	Description
suid	*smallint*	Server user ID, copied from *syslogins*.
uid	*smallint*	User ID, unique in this database, used for granting and revoking permissions. User ID 1 is "dbo".
gid	*smallint*	Group ID to which this user belongs. If *uid* = *gid*, this entry defines a group. The group "public" has *suid* = -2; all other groups have *suid*= - *gid*.
name	*sysname*	User or group name, unique in this database
environ	*varchar(255)*	Reserved

Table B-52: Columns in the sysusers *table*

Indexes

unique clustered index on *suid*
unique nonclustered index on *name*
unique nonclustered index on *uid*

Referenced by System Procedures

sp_addalias, sp_addgroup, sp_adduser, sp_changedbowner, sp_changegroup, sp_checknames, sp_checkreswords, sp_column_privileges, sp_depends, sp_dropgroup, sp_droptype, sp_dropuser, sp_helpgroup, sp_helprotect, sp_helpuser, sp_indsuspect, sp_stored_procedures, sp_table_privileges, sp_tables

C The *pubs2* Database

This is the sample database *pubs2*. The names of the 11 tables are *publishers*, *authors*, *titles*, *titleauthor*, *sales*, *salesdetail*, *stores*, *discounts*, *roysched*, *au_pix*, and *blurbs*.

The header for each column lists its datatype (including the user-defined datatypes) and its null / not null status. Defaults, rules, triggers, and indexes are noted where they apply.

Tables in the *pubs2* Database

publishers			
pub_id char(4) not null pub_idrule[a] clust, uniq	pub_name varchar(40) null	city varchar(20) null	state char(2) null
0736	New Age Books	Boston	MA
0877	Binnet & Hardley	Washington	DC
1389	Algodata Infosystems	Berkeley	CA

a. The pub_id rule states that the data must be 1389, 0736, 0877, 1622, or 1756, or must match the pattern 99[0-9][0-9].

authors								
au_id id not null clust, uniq	au_lname varchar(40) not null nonclust	au_fname varchar(20) not null	phone char(12) not null UNKNOWN[a]	address varchar(12) null	city varchar(20) null	state char(2) null	country varchar(12) null	postalcode char(10) null
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	USA	94025
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	USA	94618
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	USA	94705
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	USA	95128
274-80-9391	Straight	Dick	415 834-2919	5420 College Av.	Oakland	CA	USA	94609
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	USA	66044
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St.	Berkeley	CA	USA	94705
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	USA	95428
486-29-1786	Locksley	Chastity	415 585-4620	18 Broadway Av.	San Francisco	CA	USA	94130
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd.	Nashville	TN	USA	37215
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl.	Corvallis	OR	USA	97330
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	USA	94595
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl. #86	Ann Arbor	MI	USA	48105
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	USA	46403
724-08-9931	Stringer	Dirk	415 843-2991	5420 Telegraph Av.	Oakland	CA	USA	94609
724-80-9391	MacFeather	Stearns	415 354-7128	44 Upland Hts.	Oakland	CA	USA	94612
756-30-7391	Karsen	Livia	415 534-9219	5720 McAuley St.	Oakland	CA	USA	94609
807-91-6654	Panteley	Sylvia	301 946-8853	1956 Arlington Pl.	Rockville	MD	USA	20853
846-92-7186	Hunter	Sheryl	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	USA	95688
899-46-2035	Ringer	Anne	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152

a. The default *UNKNOWN* is inserted if no data is entered.

titles									
title_id tid not null deltitle[3] clust, uniq	title varchar(80) not null nonclust	type char(12) not null UNDECIDED[1]	pub_id char(4) null	price money null	advance money null	total_sales int null	notes varchar(200) null	pubdate datetime not null getdate()[2]	contract bit not null
BU1032	The Busy Executive's Database Guide	business	1389	19.99	5000.00	4095	An overview of available database systems with emphasis on common business applications. Illustrated.	Jun 6, 1986	1
BU1111	Cooking with Computers: Surreptitious Balance Sheets	business	1389	11.95	5000.00	3876	Helpful hints on how to use your electronic resources to the best advantage.	Jun 9, 1988	1
BU2075	You Can Combat Computer Stress!	business	0736	2.99	10125.00	18722	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations.	Jun 30, 1985	1
BU7832	Straight Talk About Computers	business	1389	19.99	5000.00	4095	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.	Jun 22, 1987	1
MC2222	Silicon Valley Gastronomic Treats	mod_cook	0877	19.99	0.00	2032	Favorite recipes for quick, easy, and elegant meals, tried and tested by people who never have time to eat, let alone cook.	Jun 9, 1989	1
MC3021	The Gourmet Microwave	mod_cook	0877	2.99	15000.00	22246	Traditional French gourmet recipes adapted for modern microwave cooking.	Jun 18, 1985	1
PC1035	But Is It User Friendly?	popular_comp	1389	22.95	7000.00	8780	A survey of software for the naive user, focusing on the 'friendliness' of each.	Jun 30, 1986	1
MC3026	The Psychology of Computer Cooking	UNDECIDED	0877	NULL	NULL	NULL	NULL	Jul 24, 1991	0
PC8888	Secrets of Silicon Valley	popular_comp	1389	20.00	8000.00	4095	Muckraking reporting by two courageous women on the world's largest computer hardware and software manufacturers.	Jun 12, 1987	1

1. The default *UNDECIDED* is inserted if no data is entered in the column.
2. The *getdate* function inserts the current date as the default if no data is entered in the column.
3. The *deltitle* trigger prohibits deleting a title if the *title_id* is listed in the *sales* table.

titles									
title_id tid not null deltitle[3] clust, uniq	title varchar(80) not null nonclust	type char(12) not null UNDECIDED[1]	pub_id char(4) null	price money null	advance money null	total_sales int null	notes varchar(200) null	pubdate datetime not null getdate()[2]	contract bit not null
PC9999	Net Etiquette	popular_comp	1389	NULL	NULL	NULL	A must-read for computer conferencing debutantes!	Jul 24, 1991	0
PS1372	Computer Phobic and Non-Phobic Individuals: Behavior Variations	psychology	0877	21.59	7000.00	375	A must for the specialist, this book examines the difference between those who hate and fear computers and those who think they are swell.	Oct 21,1990	1
PS2091	Is Anger the Enemy?	psychology	0736	10.95	2275.00	2045	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.	Jun 15, 1989	1
PS2106	Life Without Fear	psychology	0736	7.00	6000.00	111	New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. Popular audience. Sample menus included, exercise video available separately.	Oct 5, 1990	1
PS3333	Prolonged Data Deprivation: Four Case Studies	psychology	0736	19.99	2000.00	4072	What happens when the data runs dry? Searching evaluations of information-shortage effects on heavy users.	Jun 12, 1988	1
PS7777	Emotional Security: A New Algorithm	psychology	0736	7.99	4000.00	3336	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized.	Jun 12, 1988	1
TC3218	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook	0877	20.95	7000.00	375	Profusely illustrated in color, this makes a wonderful gift book for a cuisine-oriented friend.	Oct 21, 1990	1
TC4203	Fifty Years in Buckingham Palace Kitchens	trad_cook	0877	11.95	4000.00	15096	More anecdotes from the Queen's favorite cook describing life among English royalty. Recipes, techniques, tender vignettes.	Jun 12, 1985	1

1. The default *UNDECIDED* is inserted if no data is entered in the column.
2. The *getdate* function inserts the current date as the default if no data is entered in the column.
3. The *deltitle* trigger prohibits deleting a title if the *title_id* is listed in the *sales* table.

titles									
title_id tid not null deltitle[3] clust, uniq	title varchar(80) not null nonclust	type char(12) not null UNDECIDED[1]	pub_id char(4) null	price money null	advance money null	total_sales int null	notes varchar(200) null	pubdate datetime not null getdate()[2]	contract bit not null
TC7777	Sushi, Anyone?	trad_cook	0877	14.99	8000.00	4095	Detailed instructions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test.	Jun 12, 1987	1

1. The default *UNDECIDED* is inserted if no data is entered in the column.
2. The *getdate* function inserts the current date as the default if no data is entered in the column.
3. The *deltitle* trigger prohibits deleting a title if the *title_id* is listed in the *sales* table.

titleauthor			
au_id id not null nonclust	title_id tid not null nonclust	au_ord tinyint null	royaltyper int null
uniq, clust, composite			
172-32-1176	PS3333	1	100
213-46-8915	BU1032	2	40
213-46-8915	BU2075	1	100
238-95-7766	PC1035	1	100
267-41-2394	BU1111	2	40
267-41-2394	TC7777	2	30
274-80-9391	BU7832	1	100
409-56-7008	BU1032	1	60
427-17-2319	PC8888	1	50
472-27-2349	TC7777	3	30
486-29-1786	PC9999	1	100
486-29-1786	PS7777	1	100
648-92-1872	TC4203	1	100
672-71-3249	TC7777	1	40
712-45-1867	MC2222	1	100
722-51-5454	MC3021	1	75
724-80-9391	BU1111	1	60
724-80-9391	PS1372	2	25
756-30-7391	PS1372	1	75
807-91-6654	TC3218	1	100
846-92-7186	PC8888	2	50
899-46-2035	MC3021	2	25
899-46-2035	PS2091	2	50
998-72-3567	PS2091	1	50
998-72-3567	PS2106	1	100

au_pix					
au_id id not null	pic image null	format_type char(11) null	bytesize int null	pixwidth_hor char(14) null	pixwidth_vert char(14) null
409-56-7008	0x0000...	PICT	30220	626	635
486-29-1786	0x59a6...	Sunraster	27931	647	640
648-92-1872	0x59a6...	Sunraster	36974	647	640
672-71-3249	0x000a...	PICT	13487	654	639
899-46-2035	0x4949...	TIF	52023	648	641
998-72-3567	0x4949...	TIF	52336	653	637

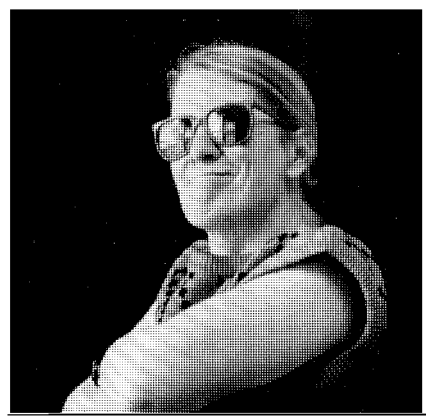
The *pic* column contains binary data, which is not reproduced in this table in its entirety. The pictures represented by this data are shown on the next page. Since the *image* data (six pictures, two each in PICT, TIF, and Sun raster file formats) is quite large, you should run the *installpix2* script **only** if you want to use or test the *image* datatype. The *image* data is supplied to show how SYBASE stores *image* data. Sybase does not supply any tools for displaying *image* data: you must use the appropriate screen graphics tools in order to display the images once you have extracted them from the database.

Authors' Portraits from the *au_pix* Table

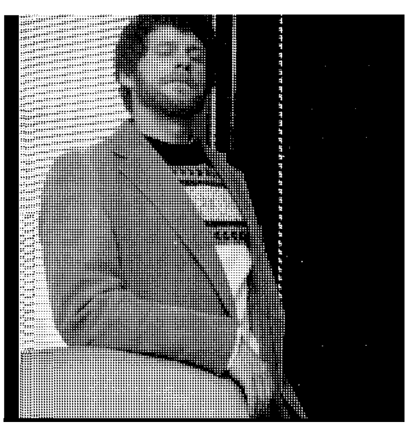
Akiko Yokomoto 672-71-3249

Chastity Locksley 486-29-1786

Anne Ringer 899-46-2035



Albert Ringer 998-72-3567



Bennet Abraham 409-56-7008

Reginald Blotchet-Halls 648-92-1872

salesdetail				
stor_id char(4) not null	ord_num varchar(20) not null	title_id tid not null title_idrule	qty smallint not null	discount float not null
nonclust		nonclust		
7896	234518	TC3218	75	40.000000
7896	234518	TC7777	75	40.000000
7131	Asoap432	TC3218	50	40.000000
7131	Asoap432	TC7777	80	40.000000
5023	XS-135-DER-432-8J2	TC3218	85	40.000000
8042	91-A-7	PS3333	90	45.000000
8042	91-A-7	TC3218	40	45.000000
8042	91-A-7	PS2106	30	45.000000
8042	91-V-7	PS2106	50	45.000000
8042	55-V-7	PS2106	31	45.000000
8042	91-A-7	MC3021	69	45.000000
5023	BS-345-DSE-860-1F2	PC1035	1000	46.700000
5023	AX-532-FED-452-2Z7	BU2075	500	46.700000
5023	AX-532-FED-452-2Z7	BU1032	200	46.700000
5023	AX-532-FED-452-2Z7	BU7832	150	46.700000
5023	AX-532-FED-452-2Z7	PS7777	125	46.700000
5023	NF-123-ADS-642-9G3	TC7777	1000	46.700000
5023	NF-123-ADS-642-9G3	BU1032	1000	46.700000
5023	NF-123-ADS-642-9G3	PC1035	750	46.700000
7131	Fsoap867	BU1032	200	46.700000
7066	BA52498	BU7832	100	46.700000
7066	BA71224	PS7777	200	46.700000
7066	BA71224	PC1035	300	46.700000
7066	BA71224	TC7777	350	46.700000
5023	ZD-123-DFG-752-9G8	PS2091	1000	46.700000
7067	NB-3.142	PS2091	200	46.700000
7067	NB-3.142	PS7777	250	46.700000
7067	NB-3.142	PS3333	345	46.700000
7067	NB-3.142	BU7832	360	46.700000
5023	XS-135-DER-432-8J2	PS2091	845	46.700000
5023	XS-135-DER-432-8J2	PS7777	581	46.700000
5023	ZZ-999-ZZZ-999-0A0	PS1372	375	46.700000
7067	NB-3.142	BU1111	175	46.700000
5023	XS-135-DER-432-8J2	BU7832	885	46.700000
5023	ZD-123-DFG-752-9G8	BU7832	900	46.700000
5023	AX-532-FED-452-2Z7	TC4203	550	46.700000
7131	Fsoap867	TC4203	350	46.700000
7896	234518	TC4203	275	46.700000
7066	BA71224	TC4203	500	46.700000

salesdetail				
stor_id char(4) not null	ord_num varchar(20) not null	title_id tid not null title_idrule	qty smallint not null	discount float not null
nonclust		nonclust		
7067	NB-3.142	TC4203	512	46.700000
7131	Fsoap867	MC3021	400	46.700000
5023	AX-532-FED-452-2Z7	PC8888	105	46.700000
5023	NF-123-ADS-642-9G3	PC8888	300	46.700000
7066	BA71224	PC8888	350	46.700000
7067	NB-3.142	PC8888	335	46.700000
7131	Asoap432	BU1111	500	46.700000
7896	234518	BU1111	340	46.700000
5023	AX-532-FED-452-2Z7	BU1111	370	46.700000
5023	ZD-123-DFG-752-9G8	PS3333	750	46.700000
8042	13-J-9	BU7832	300	51.700000
8042	13-E-7	BU2075	150	51.700000
8042	13-E-7	BU1032	300	51.700000
8042	13-E-7	PC1035	400	51.700000
8042	91-A-7	PS7777	180	51.700000
8042	13-J-9	TC4203	250	51.700000
8042	13-E-7	TC4203	226	51.700000
8042	13-E-7	MC3021	400	51.700000
8042	91-V-7	BU1111	390	51.700000
5023	AB-872-DEF-732-2Z1	MC3021	5000	50.000000
5023	NF-123-ADS-642-9G3	PC8888	2000	50.000000
5023	NF-123-ADS-642-9G3	BU2075	2000	50.000000
5023	GH-542-NAD-713-9F9	PC1035	2000	50.000000
5023	ZA-000-ASD-324-4D1	PC1035	2000	50.000000
5023	ZA-000-ASD-324-4D1	PS7777	1500	50.000000
5023	ZD-123-DFG-752-9G8	BU2075	3000	50.000000
5023	ZD-123-DFG-752-9G8	TC7777	1500	50.000000
5023	ZS-645-CAT-415-1B2	BU2075	3000	50.000000
5023	ZS-645-CAT-415-1B2	BU2075	3000	50.000000
5023	XS-135-DER-432-8J2	PS3333	2687	50.000000
5023	XS-135-DER-432-8J2	TC7777	1090	50.000000
5023	XS-135-DER-432-8J2	PC1035	2138	50.000000
5023	ZZ-999-ZZZ-999-0A0	MC2222	2032	50.000000
5023	ZZ-999-ZZZ-999-0A0	BU1111	1001	50.000000
5023	ZA-000-ASD-324-4D1	BU1111	1100	50.000000
5023	NF-123-ADS-642-9G3	BU7832	1400	50.000000
5023	BS-345-DSE-860-1F2	TC4203	2700	50.000000
5023	GH-542-NAD-713-9F9	TC4203	2500	50.000000
5023	NF-123-ADS-642-9G3	TC4203	3500	50.000000

salesdetail				
stor_id char(4) not null	ord_num varchar(20) not null	title_id tid not null title_idrule	qty smallint not null	discount float not null
nonclust		nonclust		
5023	BS-345-DSE-860-1F2	MC3021	4500	50.000000
5023	AX-532-FED-452-2Z7	MC3021	1600	50.000000
5023	NF-123-ADS-642-9G3	MC3021	2550	50.000000
5023	ZA-000-ASD-324-4D1	MC3021	3000	50.000000
5023	ZS-645-CAT-415-1B2	MC3021	3200	50.000000
5023	BS-345-DSE-860-1F2	BU2075	2200	50.000000
5023	GH-542-NAD-713-9F9	BU1032	1500	50.000000
5023	ZZ-999-ZZZ-999-0A0	PC8888	1005	50.000000
7896	124152	BU2075	42	50.500000
7896	124152	PC1035	25	50.500000
7131	Asoap132	BU2075	35	50.500000
7067	NB-1.142	PC1035	34	50.500000
7067	NB-1.142	TC4203	53	50.500000
8042	12-F-9	BU2075	30	55.500000
8042	12-F-9	BU1032	94	55.500000
7066	BA27618	BU2075	200	57.200000
7896	124152	TC4203	350	57.200000
7066	BA27618	TC4203	230	57.200000
7066	BA27618	MC3021	200	57.200000
7131	Asoap132	MC3021	137	57.200000
7067	NB-1.142	MC3021	270	57.200000
7067	NB-1.142	BU2075	230	57.200000
7131	Asoap132	BU1032	345	57.200000
7067	NB-1.142	BU1032	136	57.200000
8042	12-F-9	TC4203	300	62.200000
8042	12-F-9	MC3021	270	62.200000
8042	12-F-9	PC1035	133	62.200000
5023	AB-123-DEF-425-1Z3	TC4203	2500	60.500000
5023	AB-123-DEF-425-1Z3	BU2075	4000	60.500000
6380	342157	BU2075	200	57.200000
6380	342157	MC3021	250	57.200000
6380	356921	PS3333	200	46.700000
6380	356921	PS7777	500	46.700000
6380	356921	TC3218	125	46.700000
6380	234518	BU2075	135	46.700000
6380	234518	BU1032	320	46.700000
6380	234518	TC4203	300	46.700000
6380	234518	MC3021	400	46.700000

sales		
stor_id char(4) not null	ord_num varchar(20) not null	date datetime not null
clust, uniq		
5023	AB-123-DEF-425-1Z3	Oct 31 1985
5023	AB-872-DEF-732-2Z1	Nov 6 1985
5023	AX-532-FED-452-2Z7	Dec 1 1990
5023	BS-345-DSE-860-1F2	Dec 12 1986
5023	GH-542-NAD-713-9F9	Mar 15 1987
5023	NF-123-ADS-642-9G3	Jul 18 1987
5023	XS-135-DER-432-8J2	Mar 21 1991
5023	ZA-000-ASD-324-4D1	Jul 27 1988
5023	ZD-123-DFG-752-9G8	Mar 21 1991
5023	ZS-645-CAT-415-1B2	Mar 21 1991
5023	ZZ-999-ZZZ-999-0A0	Mar 21 1991
6380	234518	Sep 30 1987
6380	342157	Dec 13 1985
6380	356921	Feb 17 1991
7066	BA27618	Oct 12 1985
7066	BA52498	Oct 27 1987
7066	BA71224	Aug 5 1988
7067	NB-1.142	Jan 2 1987
7067	NB-3.142	Jun 13 1990
7131	Asoap132	Nov 16 1986
7131	Asoap432	Dec 20 1990
7131	Fsoap867	Sep 8 1987
7896	124152	Aug 14 1986
7896	234518	Feb 14 1991
8042	12-F-9	Jul 13 1986
8042	13-E-7	May 23 1989
8042	13-J-9	Jan 13 1988
8042	55-V-7	Mar 20 1991
8042	91-A-7	Mar 20 1991
8042	91-V-7	Mar 20 1991

stores

stor_id char(4) not null	stor_name varchar(40) null	stor_address varchar(40) null	city varchar(20) null	state char(2) null	country varchar(12) null	postalcode char(10) null	payterms varchar(12) null
7066	Barnum's	567 Pasadena Ave.	Tustin	CA	USA	92789	Net 30
7067	News & Brews	577 First St.	Los Gatos	CA	USA	96745	Net 30
7131	Doc-U-Mat: Quality Laundry and Books	24-A Avrogado Way	Remulade	WA	USA	98014	Net 60
8042	Bookbeat	679 Carson St.	Portland	OR	USA	89076	Net 30
6380	Eric the Read Books	788 Catamaugus Ave.	Seattle	WA	USA	98056	Net 60
7896	Fricative Bookshop	89 Madison St.	Fremont	CA	USA	90019	Net 60
5023	Thoreau Reading Discount Chain	20435 Walden Expressway	Concord	MA	USA	01776	Net 60

discounts

discounttype varchar(40) not null	stor_id char(4) null	lowqty smallint null	highqty smallint null	discount float not null
Initial Customer				10.5
Volume Discount		100	1000	6.7
Huge Volume Discount		1001		10
Customer Discount	8042			5

roysched			
title_id tid not null nonclust	lorange int null	hirange int null	royalty int null
BU1032	0	5000	10
BU1032	5001	50000	12
PC1035	0	2000	10
PC1035	2001	3000	12
PC1035	3001	4000	14
PC1035	4001	10000	16
PC1035	10001	50000	18
BU2075	0	1000	10
BU2075	1001	3000	12
BU2075	3001	5000	14
BU2075	5001	7000	16
BU2075	7001	10000	18
BU2075	10001	12000	20
BU2075	12001	14000	22
BU2075	14001	50000	24
PS2091	0	1000	10
PS2091	1001	5000	12
PS2091	5001	10000	14
PS2091	10001	50000	16
PS2106	0	2000	10
PS2106	2001	5000	12
PS2106	5001	10000	14
PS2106	10001	50000	16
MC3021	0	1000	10
MC3021	1001	2000	12
MC3021	2001	4000	14
MC3021	4001	6000	16
MC3021	6001	8000	18
MC3021	8001	10000	20
MC3021	10001	12000	22

roysched			
title_id tid not null nonclust	lorange int null	hirange int null	royalty int null
MC3021	12001	50000	24
TC3218	0	2000	10
TC3218	2001	4000	12
TC3218	4001	6000	14
TC3218	6001	8000	16
TC3218	8001	10000	18
TC3218	10001	12000	20
TC3218	12001	14000	22
TC3218	14001	50000	24
PC8888	0	5000	10
PC8888	5001	10000	12
PC8888	10001	15000	14
PC8888	15001	50000	16
PS7777	0	5000	10
PS7777	5001	50000	12
PS3333	0	5000	10
PS3333	5001	10000	12
PS3333	10001	15000	14
PS3333	15001	50000	16
BU1111	0	4000	10
BU1111	4001	8000	12
BU1111	8001	10000	14
BU1111	12001	16000	16
BU1111	16001	20000	18
BU1111	20001	24000	20
BU1111	24001	28000	22
BU1111	28001	50000	24
MC2222	0	2000	10
MC2222	2001	4000	12
MC2222	4001	8000	14

roysched			
title_id tid not null nonclust	lorange int null	hirange int null	royalty int null
MC2222	8001	12000	16
MC2222	8001	12000	16
MC2222	12001	20000	18
MC2222	20001	50000	20
TC7777	0	5000	10
TC7777	5001	15000	12
TC7777	15001	50000	14
TC4203	0	2000	10
TC4203	2001	8000	12
TC4203	8001	16000	14
TC4203	16001	24000	16
TC4203	24001	32000	18
TC4203	32001	40000	20
TC4203	40001	50000	22
BU7832	0	5000	10
BU7832	5001	10000	12
BU7832	10001	15000	14
BU7832	15001	20000	16
BU7832	20001	25000	18
BU7832	25001	30000	20
BU7832	30001	35000	22
BU7832	35001	50000	24
PS1372	0	10000	10
PS1372	10001	20000	12
PS1372	20001	30000	14
PS1372	30001	40000	16
PS1372	40001	50000	18

blurbs	
au_id id not null	copy text null
486-29-1786	If Chastity Locksley didn't exist, this troubled world would have created her! Not only did she master the mystic secrets of inner strength to conquer adversity when she encountered it in life, but, after "reinventing herself", as she says, by writing "Emotional Security: A New Algorithm" following the devastating loss of her cat Old Algorithm, she also founded Publish or Perish, the page-by-page, day-by-day, write-yourself-to-wellness encounter workshops franchise empire, the better to share her inspiring discoveries with us all. Her "Net Etiquette," a brilliant social treatise in its own right and a fabulous pun, is the only civilized alternative to the gross etiquette often practiced on the public networks.
648-92-1872	A chef's chef and a raconteur's raconteur, Reginald Blotchet-Halls calls London his second home. "Th' palace kitchen's me first 'ome, act'lly!" Blotchet-Halls' astounding ability to delight our palates with palace delights is matched only by his equal skill in satisfying our perpetual hunger for delicious back-stairs gossip by serving up tidbits and entrees literally fit for a king!
998-72-3567	Albert Ringer was born in a trunk to circus parents, but another kind of circus trunk played a more important role in his life years later. He grew up as an itinerant wrestler and roustabout in the reknowned Ringer Brothers and Betty and Bernie's Circus. Once known in the literary world only as Anne Ringer's wrestling brother, he became a writer while recuperating from a near-fatal injury received during a charity benefit bout with a gorilla. "Slingshotting" himself from the ring ropes, Albert flew over the gorilla's head and would have landed head first on the concrete. He was saved from certain death by Nana, an elephant he had befriended as a child, who caught him in her trunk. Nana held him so tightly that three ribs cracked and he turned blue from lack of oxygen. "I was delirious. I had an out-of-body experience! My whole life passed before me eyes. I promised myself 'If I get through this, I'll use my remaining time to share what I learned out there.' I owe it all to Nana!"
899-46-2035	Anne Ringer ran away from the circus as a child. A university creative writing professor and her family took Anne in and raised her as one of their own. In this warm and television-less setting she learned to appreciate the great classics of literature. The stream of aspiring and accomplished writers that flowed constantly through the house confirmed her repudiation of the circus family she'd been born into: "Barbarians!" The steadily growing recognition of her literary work was, to her, vindication. When her brother's brush with death brought them together after many years, she took advantage of life's crazy chance thing and broke the wall of anger that she had constructed to separate them. Together they wrote, "Is Anger the Enemy?" an even greater blockbuster than her other collaborative work, with Michel DeFrance, "The Gourmet Microwave."
672-71-3249	They asked me to write about myself and my book, so here goes: I started a restaurant called "de Gustibus" with two of my friends. We named it that because you really can't discuss taste. We're very popular with young business types because we're young business types ourselves. Whenever we tried to go out to eat in a group we always got into these long tiresome negotiations: "I just ate Italian," or "I ate Greek yesterday," or "I NEVER eat anything that's not organic!" Inefficient. Not what business needs today. So, it came to us that we needed a restaurant we could all go to every day and not eat the same thing twice in a row maybe for a year! We thought, "Hey, why make people choose one kind of restaurant over another, when what they really want is a different kind of food?" At de Gustibus you can eat Italian, Chinese, Japanese, Greek, Russian, Tasmanian, Iranian, and on and on all at the same time. You never have to choose. You can even mix and match! We just pooled our recipes, opened the doors, and never looked back. We're a big hit, what can I say? My recipes in "Sushi, Anyone?" are used at de Gustibus. They satisfy crowds for us every day. They will work for you, too. Period!
409-56-7008	Bennet was the classic too-busy executive. After discovering computer databases he now has the time to run several successful businesses and sit on three major corporate boards. Bennet also donates time to community service organizations. Miraculously, he also finds time to write and market executive-oriented in-depth computer hardware and software reviews. "I'm hyperkinetic, so being dynamic and fast-moving is a piece of cake. But being organized isn't easy for me or for anyone I know. There's just one word for that: 'databases!' Databases can cure you or kill you. If you get the right one, you can be like me. If you get the wrong one, watch out. Read my book!"

Primary and Foreign Keys in *pubs2*

Primary Keys	
Table	**Primary Key**
titles	title_id
titleauthor	au_id + title_id
authors	au_id
publishers	pub_id
roysched	title_id
sales	stor_id + ord_num
salesdetail	stor_id + ord_num
stores	stor_id
discounts	discounttype + stor_id
au_pix	au_id
blurbs	au_id

Foreign Keys		
Table	**Foreign Key**	**Primary Key Table**
titleauthor	title_id au_id	titles authors
roysched	title_id	titles
sales	title_id stor_id	titles stores
salesdetail	title_id stor_id, ord_num	titles sales
titles	pub_id	publishers
discounts	stor_id	stores
au_pix	au_id	authors
blurbs	au_id	authors

Other Objects in *pubs2*

Rules

pub_idrule

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

title_idrule

```
create rule title_idrule
as
@title_id like “BU[0-9][0-9][0-9][0-9]” or
@title_id like “[MT]C[0-9][0-9][0-9][0-9]” or
@title_id like “P[SC][0-9][0-9][0-9][0-9]” or
@title_id like “[A-Z][A-Z]xxxx” or
@title_id like “[A-Z][A-Z]yyyy”
/*valid values: BU, MC, TC, PS, PC + 4 digits or
**any two uppercase letters followed by x’s or y’s
*/
```

Defaults

typedflt

```
create default typedflt as “UNDECIDED”
```

datedflt

```
create default datedflt as getdate()
```

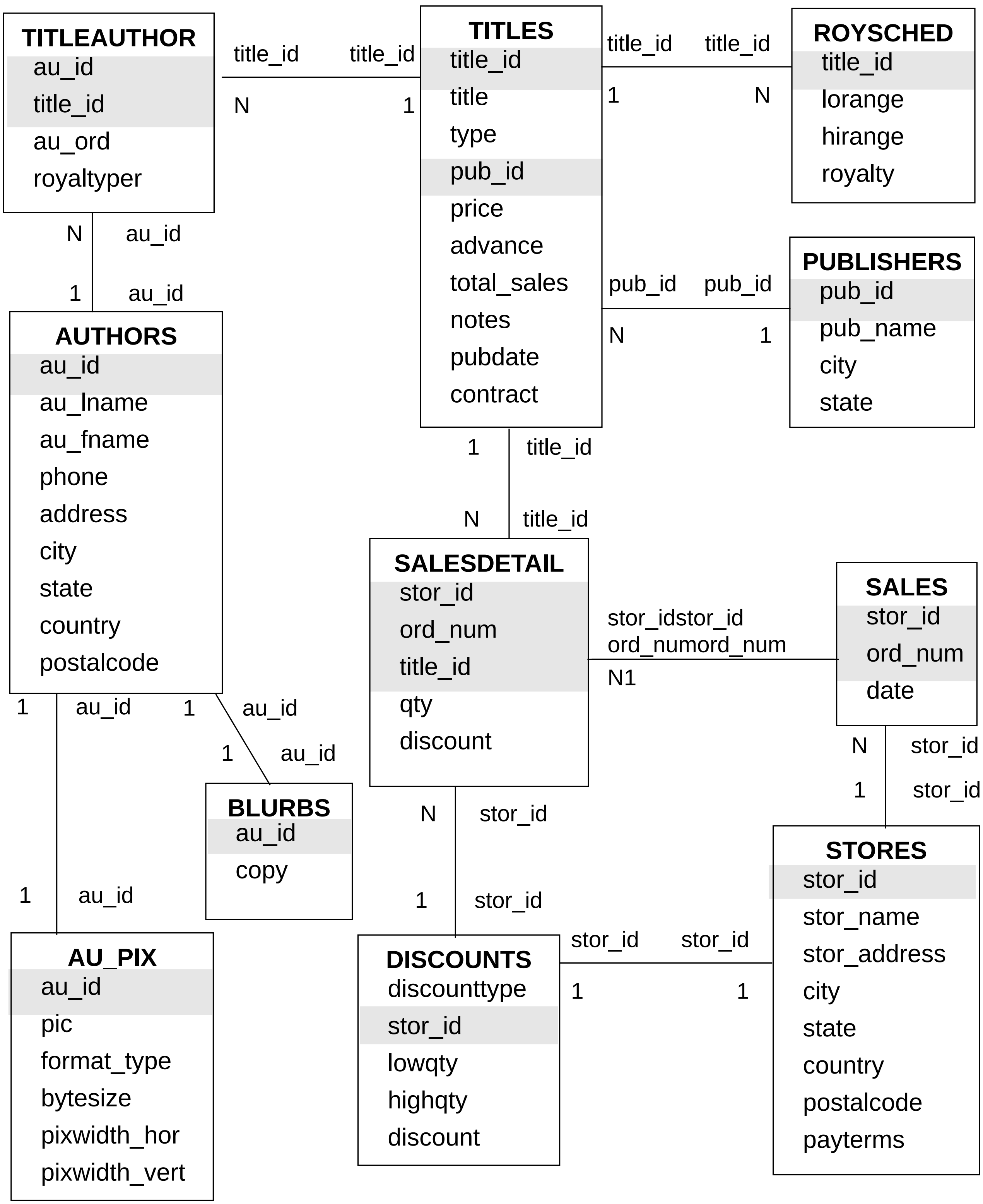
phonedflt

```
create default phonedflt as “UNKNOWN”
```

View

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

Diagram of the *pubs2* Database



D Glossary

allocation unit

An allocation unit is a logical unit of SQL Server storage equal to 256 2Kb data pages, or 1/2 megabyte. The **disk init** command initializes a new database file for SQL Server and divides it into 1/2 megabyte pieces called allocation units. On Stratus, allocation units are 256 4Kb pages, or 1 megabyte.

automatic recovery

A process that runs every time SQL Server is stopped and restarted. The process ensures that all transactions which have completed before the server went down are brought forward and all incomplete transactions are rolled back.

backup

A copy of a database or transaction log, used to recover from a media failure.

batch

One or more Transact-SQL statements terminated by an end-of-batch signal, which submits them to the SQL Server for processing. The Report Workbench and other client software supply end-of-batch signals to SQL batches automatically.

built-in functions

A wide variety of functions that take one or more parameters and return results. The built-in functions include mathematical functions, system functions, string functions, text functions, date functions, and a type conversion function.

bulk copy

The utility for copying data in and out of databases, called **bcp**.

character set

A set of specific (usually standardized) characters with an encoding scheme that uniquely defines each character. ASCII and ISO 8859-1 (Latin 1) are two common character sets.

character set conversion

Changing the encoding scheme of a set of characters on the way into or out of SQL Server. Conversion is used when SQL Server and a client communicating with it use different character sets. For example, if SQL Server uses ISO 8859-1 and a client uses Code Page 850, character set conversion must be turned on so that both server and client interpret the data passing back and forth in the same way.

checkpoint

The point at which all data pages that have been changed are guaranteed to have been written to the database device.

clustered index

An index in which the physical order and the logical (indexed) order is the same. The leaf level of a clustered index represents the data pages themselves.

code set

See **character set**.

collating sequence

See **sort order**.

command

An instruction that specifies an operation to be performed by the computer. Each command or SQL statement begins with a keyword, such as **insert**, that names the basic operation performed. Many SQL commands have one or more keyword phrases, or clauses, that tailor the command to meet a particular need.

command permissions

Permissions that apply to commands. See also **object permissions**.

command terminator

A command terminator is the end-of-batch signal that sends the batch to SQL Server for processing.

context-sensitive protection

Context-sensitive protection provides certain permissions or privileges depending on the identity of the user. This type of protection can be provided by SQL Server using a view and the *user_id* built-in function.

conversion

See **character set conversion**.

data definition

The process of setting up databases and creating database objects such as tables, indexes, rules, defaults, constraints, procedures, triggers, and views.

data dictionary

1. In SQL Server, the system tables that contain descriptions of the **database objects** and how they are structured.

2. In SQL Toolset, a tool for inspecting database objects.

data modification

Adding, deleting, or changing information in the database with the **insert**, **delete**, and **update** commands.

database

A set of related data tables and other database objects that are organized and presented to serve a specific purpose.

database device

A device dedicated to the storage of the objects that make up databases. It can be any piece of disk or a file in the file system that is used to store databases and database objects.

database object

A database object is one of the components of a database: table, view, index, procedure, trigger, column, default, constraint or rule.

Database Owner

The user who creates a database becomes the Database Owner. A Database Owner has control over all the database objects in that database. The login name for the Database Owner is “dbo”.

datatype

Specifies what kind of information each column will hold, and how the data will be stored. Datatypes include *char*, *int*, *money*, and so on. Users can construct their own datatypes in SQL Server based on the SQL Server system datatypes. User-defined datatypes are not supported in SQL Toolset.

date function

A function that displays information about dates and times, or manipulates date or time values. The five date functions are **getdate, datename, datepart, datediff** and **dateadd.**

dbo

In a user’s own database, SQL Server recognizes the user as *dbo*. A database owner logs into SQL Server using his or her assigned login name and password.

deadlock

A situation which arises when two users, each having a **lock** on one piece of data, attempt to acquire a lock on the other’s piece of data. The SQL Server detects deadlocks, and kills one user’s process.

default

The option chosen by the system when no other option is specified.

default database

The database that a user gets by default when he or she logs in.

default language

1. The default language of a user is the language that displays that user's prompts and messages. It can be set with **sp_modifylogin** or the **language** option of the **set** command.

2. The SQL Server default language is the language that is used to display prompts and messages for all users unless a user chooses a different language.

demand lock

A demand lock prevents any more shared locks from being set on a data resource (table or data page). Any new shared lock request has to wait for the demand lock request to finish.

dirty read

"Dirty reads" occur when one transaction modifies a row, and then a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid.

disk allocation pieces

Disk allocation pieces are the groups of allocation units from which SQL Server constructs a new database file. The minimum size for a disk allocation piece is one allocation unit, or 256 2Kb pages (256 4Kb pages on Stratus).

disk initialization

The process of preparing a database partition, foreign device or file for SQL Server use. Once the device is initialized, it can be used for storing databases and database objects. The command used to initialize a database device is **disk init**.

disk mirror

A duplicate of a SQL Server **database device**. All writes to the device being mirrored are copied to a separate physical device, making the second device an exact copy of the device being mirrored. If one of the devices fails, the other contains an up-to-date copy of all transactions. The command **disk mirror** starts the disk mirroring process.

display precision

The number of significant binary digits offered by the default display format for *real* and *float* values. Internally, *real* and *float* values are stored with a precision less than or equal to that of the platform-specific datatypes on which they are built. For display purposes, Sybase reals have 9 digits of precision; Sybase floats, 17.

dump striping

Interleaving of dump data across several **dump volumes**.

dump volume

A single tape, partition or file used for a database or transaction dump. A dump can span many volumes, or many dumps can be made to a single tape volume.

dynamic dump

A dump made while the database is active.

engine

A process running a SQL Server that communicates with other server processes using shared memory. An engine can be thought of as one CPU's worth of processing power. It does not represent a particular CPU on a machine. Also referred to as "server engine."A SQL Server running on a uniprocessor machine will always have one engine, engine 0. A SQL Server running on a multiprocessor machine can have one or more engines. The maximum number of engines running on SQL Server can be reconfigured using the **max online engines** configuration variable.

error message

A message that SQL Server issues, usually to the user's terminal, when it detects an error condition.

error state number

The number attached to an SQL Server error message that allows unique identification of the line of SQL Server code at which the error was raised.

exclusive locks

Locks which prevent any other transaction from acquiring a lock until the original lock is released at the end of a transaction, always applied for update (**insert**, **update**, **delete**) operations.

expression

1. In SQL Server, returns values. An expression can be a computation, column data, a built-in function, or a subquery.

2. A basic unit of the APT-SQL language, composed of operators and other expressions.

format file

A file created while using **bcp** to copy data out from a table in a SQL Server database to an operating system file. The format file contains information on how the data being copied out is formatted and can be used to copy the data back into a SQL Server table or to perform additional copy outs.

free-space threshold

A user-specified threshold that specifies the amount of space on a segment, and the action to be taken when the amount of space available on that segment is less than the specified space.

functions

See built-in functions.

global variable

1. In APT-SQL, a variable declared with **global variable**. The value of a global variable is available to any APT-SQL routine that declares it.

2. In SQL Server, global variables are system-defined variables that SQL Server updates on an ongoing basis. For example, *@@error* contains the last error number generated by the system.

guest

If the user name "guest" exists in the *sysusers* table of a database, any user with a valid SQL Server login can use that database, with limited privileges.

hexadecimal string

A hexadecimal-encoded binary string that begins with the prefix 0x and can include the digits 0 through 9 and the upper- and lowercase letters A through F. The interpretation of hexadecimal strings is platform specific. For some systems, the first byte after the prefix is the most significant; for others, the last byte is most significant. For example, the string 0x0100 is interpreted as 1 on some systems and as 256 on others.

identifier

1. A string of characters used to identify a database object, such as a table name or column name.

2. In APT-SQL, any string of characters which identifies an object in a routine. Includes, but is not limited to: variables, fields, groups, menu actions, forms, sqlrows, channels, and routine names.

initialization

See **disk initialization**.

int

A signed 32 bit integer value.

intent lock

An intent lock indicates the intention to acquire a shared or exclusive lock on a data page.

isolation level

Also called "locking level," "isolation level" specifies the kinds of actions which are not permitted while the current transaction executes. The ANSI standard defines 3 levels of isolation for SQL transactions. Level 1 prevents **dirty reads**, and level 2 also prevents **non-repeatable reads**. Level 3 prevents both types of reads and **phantoms**; it is equivalent to doing all **select**s with **holdlock**. The user controls the isolation level with the **set** option **transaction isolation level**; the default is isolation level 1.

kernel

A module within SQL Server that acts as the interface between SQL Server and the operating system.

keyword

A word or phrase that is reserved for exclusive use by Transact-SQL. Also known as reserved word.

last-chance threshold

A default threshold in SQL Server that suspends or kills user processes if the transaction log has run out of room. This threshold leaves just enough space for the de-allocation records for the pages cleared by the log dump itself. Threshold events calls a user-defined procedure, the default name is **sp_thresholdaction.** This procedure is **not** supplied by Sybase, it must be written by the System Administrator.

leaf level

The bottom level of a clustered or non-clustered index. In a clustered index the leaf level contains the actual data pages of the table.

livelock

A request for an **exclusive lock** that is repeatedly denied because a series of overlapping **shared locks** keeps interfering. The SQL Server detects the situation after four denials, and refuses further shared locks.

locking

The process of restricting access to resources in a multi-user environment to maintain security and prevent concurrent access problems. SQL Server automatically applies locks to tables or pages.

locking level

See **isolation level**.

login

The name a user uses to log in to SQL Server. A login is valid if SQL Server has an entry for that user in the system table *syslogins*.

Master Database

Controls the user databases and the operation of SQL Server as a whole. Known as *master*, it keeps track of such things as user accounts, ongoing processes, and system error messages.

message number

The number that uniquely identifies an error message.

mirror

See **disk mirror**.

model database

A template for new user databases. The **buildmaster** program and the **installmodel** script create *model* when SQL Server is installed. Each time the **create database** command is issued, SQL Server makes a copy of *model* and extends it to the size requested, if necessary.

multibyte character set

A character set that includes characters encoded using more than one byte. EUC JIS and Shift-JIS are examples of character sets that include several types of characters represented by multiple bytes in a Japanese language environment.

non-clustered index

An index that stores key values and pointers to data. The leaf level points to data pages rather than containing the data itself. Compare to **clustered index**.

non-repeatable read

"Non-repeatable reads" occur when one transaction reads a row and then a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read.

normalization rules

The standard rules of database design in a relational database management system.

null

Having no explicitly assigned value. NULL is not equivalent to zero, or to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another value of NULL.

object permissions

Permissions that regulate the use of certain commands (data modification commands, plus **select, truncate table** and **execute**) to specific tables, views, columns or procedures. See also **command permissions**.

objects

See database objects.

operating system

A group of programs that translates your commands to the computer, helping you perform such tasks as creating files, running programs, and printing documents.

parameter

1. An argument to a stored procedure.

2. A value passed between routines and/or forms.

permission

The authority to perform certain actions on certain database objects or to run certain commands.

phantoms

"Phantoms" occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, **update**, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.

precision

A positive integer that determines the maximum number of digits that can be represented in a *decimal*, *numeric*, or *float* column.

privilege

See **permission**.

query

1. A request for the retrieval of data with a **select** statement.

2. Any SQL statement that manipulates data.

recovery

The process of rebuilding one or more databases from database dumps and log dumps. *See also* **automatic recovery**.

remote procedure calls

1. A **stored procedure** executed on a different SQL Server from the server the user is logged into.

2. In APT-SQL, remote procedure calls are stored procedures executed with the **remote** statement. Stored procedures executed with **remote** can be on a remote server, or on the local server.

sa

See **System Administrator**.

scale

A nonnegative integer that determines the maximum number of digits that can be represented to the right of the decimal point. The scale of a datatype cannot be greater than its **precision**.

schema

A "schema" is a persistent object in the database. It consists of the collection of objects associated with a particular schema name and user authorization identifier. The objects are tables, views, domains, constraints, assertions, privileges and so on. A schema is created by a **create schema** statement.

segment

A named subset of database devices available to a particular database. It is a label that points to one or more database devices. Segments can be used to control the placement of tables and indexes on specific database devices.

server engine

See **engine**.

server user ID

The ID number by which a user is known to SQL Server.

severity level number

The severity of an error condition: errors with severity levels of 19 and above are fatal errors.

shared lock

A **lock** created by non-update ("read") operations. Other users may read the data concurrently, but no transaction can acquire an **exclusive** lock on the data until all the shared locks have been released.

sort order

Used by SQL Server to determine the order in which character data is sorted. Also called **collating sequence**.

SQL Server

The server in Sybase's "Client-Server" architecture. SQL Server manages multiple databases and multiple users, keeps track of the actual location of data on disks, maintains mapping of logical data description to physical data storage, and maintains data and procedure caches in memory.

statement

A statement begins with a keyword that names the basic operation or command to be performed.

stored procedure

A collection of SQL statements and optional control-of-flow statements stored under a name. SQL Server-supplied stored procedures are called **system procedures**.

System Administrator

A user in charge of SQL Server system administration, including creating user accounts, assigning permissions, and creating new databases. At installation time, the System Administrator's login name is *sa*. The *sa* user can assign the System Administrator role to other logins, providing greater accountability.

system databases

The four databases on a newly installed SQL Server: the Master Database (*master*), which controls user databases and the operation of the SQL Server; the Temporary Database (*tempdb*), used for temporary tables; the System Procedures Database (*sybsystemprocs*) and the Model Database (*model*) which is used as a template to create new user databases.

system function

A function that returns special information from the database, particularly from the system tables.

system procedures

Stored procedures that SQL Server supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from the system tables, or mechanisms for accomplishing database administration and other tasks that involve updating system tables.

system table

One of the data dictionary tables. The system tables keep track of information about the SQL Server as a whole and about each user database. The Master Database contains some system tables that are not in user databases.

temporary database

The temporary database in SQL Server, *tempdb*, that provides a storage area for temporary tables and other temporary working storage needs (for example, intermediate results of **group by** and **order by**).

transaction

A mechanism for ensuring that a set of actions is treated as a single unit of work.

transaction log

A system table (*syslogs*) in which all changes to the database are recorded.

trigger

A special form of **stored procedure** that goes into effect when a user gives a change command such as **insert**, **delete**, or **update** to a specified table or column. Triggers are often used to enforce referential integrity.

user authorization identifier

"User authorization identifiers" are associated with each schema. All the objects are said to be owned by or to have been created by the associated user authorization identifier for the schema.

user id

The ID number by which a user is known in a specific database. Distinct from **server user ID.**

view

An alternative way of looking at the data in one or more tables. Usually created as a subset of columns from one or more tables.

Index

The index is divided into three sections:

- Symbols

  Indexes each of the symbols used in SYBASE SQL Server documentation.

- Numbers

  Indexes entries which begin numerically.

- Subjects

  Indexes subjects alphabetically.

Page numbers in **bold** are primary references.

Symbols

Numerics

A

B

C

D

E

F

G

J

K

L

M

O

S

T

U

V

W

X