Sybase® SQL Server™
System Administration Guide

Sybase SQL Server Release 11.0.x

Document ID: 32500-01-1100-03

Last Revised: February 7, 1996

Principal author: Server Publications Group

Document ID: 32500-01-1100

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

Document Orders

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

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

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

Restricted Rights

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

Table of Contents

About This Book

1. Overview of System Administration

2. System Databases

3. System Administration for Beginners

4. Diagnosing System Problems

5. Overview of Disk Resource Issues

6. Initializing Database Devices

7. Mirroring Database Devices

8. Configuring Memory

9. Configuring Data Caches

10. Managing Multiprocessor Servers

11. Setting Configuration Parameters

12. Configuring Character Sets, Sort Orders, and Message Language

13. Converting Character Sets Between SQL Server and Clients

14. Creating User Databases

15. Setting Database Options

16. Creating and Using Segments

17. Checking Database Consistency

18. Developing a Backup and Recovery Plan

19. Backing Up and Restoring User Databases

20. Backing Up and Restoring the System Databases

21. Managing Free Space with Thresholds

Index

List of Figures

List of Tables

About This Book

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 System Administrators and Database Owners.

How to Use This Book

This manual contains five sections. Part 1, “Introduction,” describes basic system administration issues:

- Chapter 1, “Overview of System Administration,” describes the structure of the Sybase system.
- Chapter 2, “System Databases,” discusses the contents and function of the SQL Server system databases.
- Chapter 3, “System Administration for Beginners,” summarizes important tasks that new System Administrators need to perform.
- Chapter 4, “Diagnosing System Problems,” discusses SQL Server and Backup Server™ error handling and shows how to shut down servers and kill user processes.

Part 2, “Managing Physical Resources,” describes how to set up and use disks, memory, and processors with SQL Server:

- Chapter 5, “Overview of Disk Resource Issues,” provides an overview of SQL Server disk resource issues.
- Chapter 6, “Initializing Database Devices,” describes how to initialize and use database devices.
- Chapter 7, “Mirroring Database Devices,” describes how to mirror database devices for nonstop recovery from media failures.
- Chapter 8, “Configuring Memory,” explains how to configure SQL Server to use the available memory on your system.

- Chapter 9, “Configuring Data Caches,” discusses how to create named caches in memory and bind objects to those caches.
- Chapter 10, “Managing Multiprocessor Servers,” explains how to use multiple CPUs with SQL Server and discusses system administration issues that are unique to symmetric multiprocessing (SMP) environments.

Part 3, “Configuring SQL Server Behavior,” explains how to configure and use different SQL Server features:

- Chapter 11, “Setting Configuration Parameters,” summarizes the **sp_configure** parameters, which control many aspects of SQL Server behavior.
- Chapter 12, “Configuring Character Sets, Sort Orders, and Message Language,” discusses international issues, such as the files included in the Language Modules and how to configure a SQL Server language, sort order, and character set.
- Chapter 13, “Converting Character Sets Between SQL Server and Clients,” discusses character set conversion between SQL Server and clients in a heterogeneous environment.

Part 4, “Managing Databases and Database Objects,” describes how to create and administer databases and segments:

- Chapter 14, “Creating User Databases,” discusses the physical placement of databases, tables, and indexes, and the allocation of space to them.
- Chapter 15, “Setting Database Options,” describes how to set database options.
- Chapter 16, “Creating and Using Segments,” describes how to use segments, which are named collections of database devices, in databases.
- Chapter 17, “Checking Database Consistency,” describes how to use the database consistency checker, **dbcc**, to detect and fix database problems.

Part 5, “Backup and Recovery,” describes how to develop and execute a backup and recovery plan for the SQL Server system:

- Chapter 18, “Developing a Backup and Recovery Plan,” discusses the capabilities of the Backup Server and how to develop your backup strategy.
- Chapter 19, “Backing Up and Restoring User Databases,” discusses how to recover user databases.

- Chapter 20, “Backing Up and Restoring the System Databases,” discusses how to recover system databases.
- Chapter 21, “Managing Free Space with Thresholds,” discusses managing space with thresholds.

Related Documents

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

Other manuals you may find useful are:

- The SQL Server installation and configuration guide for your platform, which describes the installation procedures for SQL Server and the operating system-specific system administration tasks.
- *SQL Server Performance and Tuning Guide*, which explains how to tune SQL Server for maximum performance. The book includes information about database design issues that affect performance, query optimization, how to tune SQL Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.
- *SQL Server Reference Manual*, which contains detailed information on all of the commands and system procedures discussed in this manual.
- *SQL Server Reference Supplement*, which contains a list of the Transact-SQL® reserved words, definitions of system tables, a description of the *pubs2* sample database, a list of SQL Server error messages, and other reference information that is common to all the manuals.
- *SQL Server Security Administration Guide*, which explains how to use the security features provided by SQL Server to control user access to data. The manual includes information about how to add users to SQL Server, how to give them controlled access to database objects and procedures, and how to manage remote SQL Servers.

- *SQL Server Security Features User’s Guide*, which explains how to use the security features of SQL Server.
- SQL Server utility programs manual, which documents the Sybase utility programs such as **isql** and **bcp**, which are executed at the operating system level.
- *Transact-SQL User’s Guide*, which documents Transact-SQL, Sybase’s enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system.
- *What’s New in Sybase SQL Server Release 11.0?*, which describes the new features in SQL Server release 11.0.

Conventions Used in This Manual

The following sections describe the style conventions used in this manual.

Formatting SQL Statements

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

SQL Syntax Conventions

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

Table 1: Syntax statement conventions

Key	Definition
`command`	Command names, command option names, utility names, utility flags, and other keywords are in **`bold Courier`** in syntax statements, and in **bold Helvetica** in paragraph text.
`variable`	Variables, or words that stand for values that you fill in, are in italics.

Table 1: Syntax statement conventions (continued)

Key	Definition
`{ }`	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.
`[ ]`	Square brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.
`( )`	Parentheses are to be typed as part of the command.
`|`	The vertical bar means you may select only one of the options shown.
`,`	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

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

  sp_dropdevice [*device_name*]

  or, for a command with more options:

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

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

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

  ```
  select * from publishers
  ```

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

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

  (3 rows affected)
  ```

Case

You can disregard case when you type keywords:

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

Obligatory Options {You Must Choose At Least One}

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

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

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

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

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

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

  ```
  [anchovies]
  ```

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

  ```
  [beans | rice | sweet_potatoes]
  ```

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

  ```
  [extra_cheese, avocados, sour_cream]
  ```

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

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

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

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

An ellipsis can also be used inline to signify portions of a command that are left out of a text example. The following syntax statement represents the complete **create database** command, even though required keywords and other options are missing:

```
create database...for load
```

Expressions

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

Table 2: Types of expressions used in syntax statements

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

If You Need Help

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

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

Introduction

1 Overview of System Administration

This chapter introduces the basic topics of SQL Server system administration. It includes these sections:

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
- Configuring 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 SQL Server, make recommendations about standardizing data definitions across applications, and so on.

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). Both System Administrators and application designers will read the *Performance and Tuning Guide*.

The *System Administration Guide* (this book) addresses physical storage issues, backup and recovery, configuring SQL Server, and so on. The *Security Administration Guide* describes how to add users to SQL Server, how to give them controlled access to database objects and procedures, how to manage remote SQL Servers, and how to administer auditing. Functions specific to an application—data definition and maintaining referential integrity—are discussed in other manuals. Installation of SQL Server is discussed in the platform-specific SQL Server installation and configuration guide.

Roles Required for System Administration Tasks

Many of the commands and procedures discussed in this manual require the System Administrator or System Security Officer role. Other sections in this manual are relevant to Database Owners. A Database Owner's user name 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 he or she is using the database.

The *Security Administration Guide* contains a complete description of the roles supported by SQL Server: System Administrator, System Security Officer, and Operator.

In addition, the *Security Administration Guide* describes the special status of the owners of databases and other objects.

Using *isql* to Perform System Administration Tasks

Nearly all the system administration tasks described in this guide require that you, as a System Administrator, connect to SQL Server with the utility **isql**. This section provides some basic information about using **isql**. For more information about **isql**, see the SQL Server utility programs manual.

Starting *isql*

To start **isql** on most platforms, type this command at an operating system prompt:

```
isql -Uusername
```

where *username* is the user name of the System Administrator. SQL Server prompts you for your password.

➤ ***Note***

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

You can use **isql** in command-line mode to enter many of the Transact-SQL examples in this manual.

Entering Statements

The statements that you enter in **isql** can span several lines. **isql** does not process statements until you type “go” on a separate line. For example:

```
1> select *
2> from sysobjects
3> where type = "TR"
4> go
```

The examples printed in this manual do not include the **go** command between statements. If you are typing the examples, you must enter the **go** command to see the sample output.

Saving and Reusing Statements

This manual frequently asks you to save the Transact-SQL statements you use to create or modify user databases and database objects. The easiest way to accomplish this is to create or copy the statements to an ASCII-formatted file. You can then use the file to supply statements to **isql** if you need to re-create databases or database objects later.

The syntax for using **isql** with an ASCII-formatted file is:

```
isql -Uusername -Ifilename
```

where *filename* is the full path and file name of the file that contains Transact-SQL statements. On UNIX and other platforms, use the less-than symbol (**<**) to redirect the file.

The Transact-SQL statements in the ASCII file must use valid syntax and the **go** command.

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 the *SQL Server Reference Supplement*. Some of the system tables are discussed in detail in later chapters of this manual.

Querying the System Tables

You can query the system tables just like any other tables. For example, the following statement 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:

Table 1-1: Procedures that query the system tables

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

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

Keys in System Tables

Primary, foreign, and common keys for the system tables are defined in the *master* and *model* databases. You can get a report on defined keys 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 *SQL Server System Tables Diagram* included with SQL Server shows the relationships between columns in the system tables.

Warnings About System Tables

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

Update system tables only when you are instructed to do so by Technical Support or by an instruction in the *SQL Server Troubleshooting Guide* or in this manual.

When you 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 can 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, issue a **commit transaction** command.
- Do not create triggers on system tables.

◆ ***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 a user session.

System Procedures

The names of all 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 system procedures are discussed in the *Security Administration Guide* and the *SQL Server Reference Manual*.

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 compromise 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 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_helpprotect**, and **sp_lock**.

The *spt_values* table can be updated only by an upgrade; you should never modify it. To see how it is used, execute **sp_helptext** and 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 the sections where they are relevant. The *SQL Server Reference Manual* 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 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 to system tables** configuration parameter. Any stored procedure created while this parameter is set to “on” will **always** be able to update system tables, even when **allow updates to system tables** is set to “off.” To create a stored procedure that updates the system tables:

1. **Use sp_configure** to set **allow updates to system tables** to “on.”
2. Create the stored procedure with the **create procedure** command.
3. Use **sp_configure** to set **allow updates to system tables** to “off.”

◆ ***WARNING!***

Use extreme caution when you modify system tables. Always test the procedures that modify system tables in development or test databases, not in your production database.

Logging Error Messages

SQL Server writes start-up information to a local error log file each time it boots. The name and location of the error log file is determined by a start-up parameter (-e for most platforms). The installation program automatically sets the error log location when you configure a new SQL Server. See the SQL Server installation and configuration guide to learn the default location and file name.

Many error messages from SQL Server go to the user’s terminal only. However, fatal error messages (severity levels 19 and above), kernel error messages, and informational messages from SQL Server are recorded in the error log file.

SQL Server keeps the error log file open until you stop the server process. If you need to reduce the size of the error log by deleting old messages, stop the SQL Server process before you do so.

➤ ***Note***

On some platforms such as Windows NT, SQL Server also records error messages in the operating system event log. See the SQL Server installation and configuration guide for additional information about error logs.

The Interfaces File

SQL Server can communicate with other SQL Servers, Open Server applications, and client software on the network. Clients can talk to one or more servers, and servers can communicate with other servers via remote procedure calls. In order for products to interact with one another, each needs to know where the others reside on the network. This information is stored in an interfaces file (usually named *interfaces*, *interfac*, or *sql.ini*, depending on the operating system).

The interfaces file is like an address book. It lists the name and address of every known server. When you use a client program to connect to a server, the program looks up the server name in the interfaces file and then connects to the server using the address, as shown in Figure 1-1.

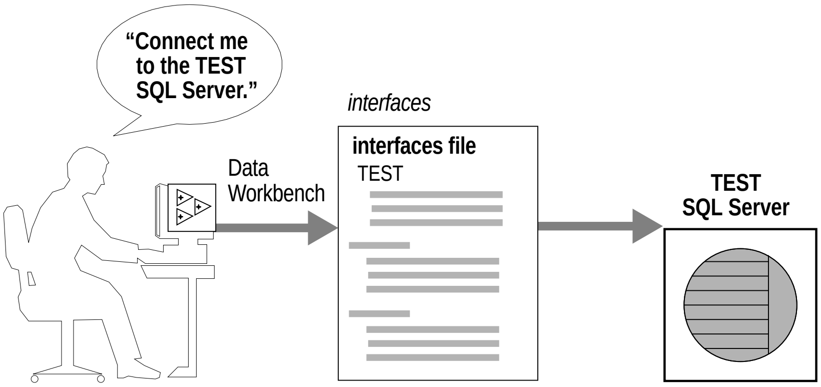


Figure 1-1: Connecting to SQL Server

The name, location, and contents of the interfaces file differ between operating systems. Also, the format of the SQL Server addresses in the interfaces file differs between network protocols.

When you install SQL Server, the installation program creates a simple interfaces file that you can use for local connections to SQL Server over one or more network protocols. As a System Administrator, it is your responsibility to modify the interfaces file and distribute it to users so that they can connect to SQL Server over the network. See the SQL Server installation and configuration guide for information about the interfaces file for your platform.

2 System Databases

This chapter describes the system databases that reside on all SQL Server systems. It also describes optional Sybase-supplied databases that you can install. This chapter includes the following sections:

Overview of System Databases

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

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

Optionally, you can install:

- The auditing database, *sybsecurity*. Use the installation program to install this database.
- The sample database, *pubs2*. Use **isql** and the installation script for *pubs2* (named **installpubs2** on most platforms) to install this database.
- The syntax database, *sybsyntax*. Use **isql** and several installation scripts to install this database.

The *master, model*, and temporary databases 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.

◆ ***WARNING!***

Do not store user databases on the master device. Storing user databases on *d_master* makes it difficult to recover the system databases if they become damaged. Also, you may not be able to recover user databases stored on *d_master* using the instructions in Chapter 20, "Backing Up and Restoring the System Databases."

The *sybsecurity* database should be installed on its own device and segment. This is discussed in the SQL Server installation and configuration guide.

The *sybsystemprocs* database can be installed on a device of your choice. You may want to modify the installation scripts for *pubs2* and the *sybsyntax* to share the device you create for *sybsystemprocs*.

➤ ***Note***

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

master Database

The *master* database controls the operation of SQL Server as a whole and stores information about all user databases and their associated database devices. 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*)
- Configurable environment variables (in *sysconfigures*)
- System error messages (in *sysmessages*)
- Databases on SQL Server (in *sysdatabases*)
- Storage space allocated to each database (in *sysusages*)
- Tapes and disks mounted on the system (in *sysdevices*)

- 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 that are online (in *sysengines*)

Because *master* stores information about user databases and devices, you must be in the *master* database in order to issue the **create database**, **alter database**, **disk init**, **disk refit**, **disk reinit**, and 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. 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.

◆ ***WARNING!***

Never place user objects in *master*. Storing user objects in *master* can cause the transaction log to fill quickly. If the transaction log runs out of space completely, you will not be able to use dump transaction commands to free space in *master*.

Another way to discourage users from creating objects in *master* is to change the default database for users (the database to which a user is connected when he or she logs in) with the system procedure **sp_modifylogin**. This procedure is discussed in the *Security Administration Guide*.

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

Protecting SQL Server by Backing Up *master*

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

- Performing frequent backups of the *master* database and all 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 the *master* database. See Chapter 20, “Backing Up and Restoring the System Databases.”

- Keeping a copy (preferably offline) 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 can use the recovery procedures described in Chapter 20, “Backing Up and Restoring the System Databases.” If you do not have current copies of that information, it will be much more difficult to recover SQL Server when the *master* database is damaged.

model Database

SQL Server includes the *model* database, which provides a template, or prototype, for new user databases. Each time a user enters the **create database** command, SQL Server makes a copy of the *model* database and extends the new database to the size specified by the **create database** command.

➤ ***Note***

A new database cannot be smaller than the *model* database.

The *model* database contains the required system tables for each user database. You can modify *model* to customize the structure of newly created databases—everything you do to *model* will be reflected in each new database. Some of the changes that System Administrators commonly make to *model* are:

- Adding user-defined datatypes, rules, or defaults.

- Adding users who should have access to all databases on SQL Server.
- Granting default privileges, particularly for guest accounts.
- Setting database options such as **select into/bulkcopy**. The settings will be reflected in all new databases. Their original value in *model* is "off". Chapter 15, "Setting Database Options," describes the database options.

Typically, most users do not have permission to modify the *model* database. There is not much point in granting read permission either, since SQL Server copies its entire contents 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 the user's current database. If there is no procedure there with that name, SQL Server looks for it in *sybsystemprocs*. If there is 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 back up 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.

The default size of *tempdb* is 2MB. 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 the *tempdb* 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. Space can be added from the master device or from any other database device.

Creating Temporary Tables

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.

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 nonsharable 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 restart 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 restart SQL Server, it copies *model* to *tempdb,* which clears the database. Temporary tables are not recoverable.

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 other default system tables that are derived from *model*.

The audit system is discussed in more detail in the *Security Administration Guide*.

pubs2 Sample 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 documentation. See the SQL Server installation and configuration guide for information about installing *pubs2*.

Maintaining the Sample Database

The sample database contains a guest user mechanism that allows access to the database by any authorized SQL Server. The guest user has been given a wide range of privileges in *pubs2*, including permissions to select, insert, update, and delete user tables. For more information about the guest user mechanism and a list of the guest permissions in *pubs2*, see the *Security Administration Guide*.

The *pubs2* database requires at least 2MB. If possible, you should give each new user a clean copy of *pubs2* so that she or he is not confused by other users' changes. If you want to place *pubs2* on a specific database device, edit the installation script before installing the database.

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

pubs2 image Data

SQL Server includes a script for installing *image* data in the *pubs2* database. The *image* data consists of six pictures, two each in PICT, TIF, and Sun raster file formats. Sybase does not provide any tools for displaying *image* data. You must use the appropriate screen graphics tools to display the images after you extract them from the database.

See the SQL Server installation and configuration guide for your platform for information about installing the *image* data in *pubs2*.

sybsyntax Database

The syntax database, *sybsyntax*, contains syntax help for Transact-SQL commands, Sybase system procedures, SQL Server utilities, and Open Client DB-Library™ routines. Users can retrieve this information using the system procedure **sp_syntax.** For example, to learn the syntax of the Transact-SQL **select** command, type:

```
sp_syntax "select"
```

SQL Server includes two scripts for creating the *sybsyntax* database. The first script, usually named *ins_syn_sql*, installs syntax help for Transact-SQL commands, Sybase system procedures, and SQL Server utilities. The second script, usually named *ins_syn_dblib*, installs syntax help for Open Client DB-Library routines. You can install one or both of these scripts.

See the SQL Server installation and configuration guide for instructions on how to install *sybsyntax*. See the *SQL Server Reference Manual* for more information about **sp_syntax**.

3 System Administration for Beginners

This chapter outlines some of the basic tasks that System Administrators perform over the life of a SQL Server. It includes the following sections:

This chapter is primarily intended to:

- Introduce new System Administrators to important topics
- Help System Administrators find information in the Sybase documentation

Experienced administrators may also find this chapter useful for organizing their ongoing maintenance activities.

You can find more information about the tasks presented in this chapter in other SQL Server manuals or in other Sybase documentation; references to individual books and chapters are included where applicable.

Using “Test” Servers

Whether you are a new or experienced System Administrator, it is always best to install and use a “test” and / or “development” SQL Server, with the intention of removing it before creating the “production” server. In general, using a test server makes it easier to plan and test different configurations and less stressful to recover from mistakes. It is much easier to learn how to install and administer new features when there is no risk of having to restart a production server or re-create a production database.

If you decide to use a test server, it is always best to do so from the point of installing or upgrading SQL Server through the process of configuring the server. It is in these steps that you make some of the most important decisions about your final production system. The following sections describe the ways in which using a test server can help System Administrators.

Understanding New Procedures and Features

Using a test server allows you to practice basic administration procedures before performing them in a production environment. If you are a new SQL Server administrator, many of the procedures discussed in this book may be unfamiliar to you, and it may take several attempts to complete a task successfully. However, even experienced administrators will benefit from practicing techniques that are introduced by new features in SQL Server release 11.0.

Planning Resources

While configuring a new server, administrators frequently discover that additional disks, memory, or hardware are needed to support their performance and throughput goals. Working with a test server helps you plan the final resource requirements for your system and helps you discover resource deficiencies that you might not have anticipated.

In particular, disk resources can have a dramatic effect on the final design of the production system. For example, you may decide that a particular database requires nonstop recovery in the event of a media failure. This would necessitate configuring one or more additional database devices to mirror the critical database. Discovering these resource requirements in a test server allows you to change the physical layout of databases and tables without affecting database users.

You can also use a test server as a “safe” environment for benchmarking both SQL Server and your applications using different hardware configurations. This allows you to determine the optimal setup for physical resources at both the SQL Server level and the operating system level before bringing the entire system online for general use.

Achieving Performance Goals

Although you can sometimes improve performance “on the fly” with production servers, most performance objectives can be met only by carefully planning a database's design and configuration. For example, you may discover that the insert and I/O performance of a particular table is a bottleneck. In this case, the best course of action may be to re-create the table on a dedicated segment and partition the table. Changes of this nature are disruptive to a production system, and even changing a configuration parameter can require you to reboot SQL Server.

Installing Sybase Products

The responsibility for installing SQL Server and other Sybase products is sometimes placed with the System Administrator. If installation is one of your responsibilities, use the following pointers to help you in the process.

Check Product Compatibility

Before installing new products or upgrading existing products, always read the *Release Bulletin* included with the products to understand any compatibility issues that might affect your system. Compatibility problems can occur between hardware and software and between the same software of different release levels. Reading the *Release Bulletin* in advance can save the time and guesswork of troubleshooting known compatibility problems.

Also, refer to the lists of known problems that are installed with SQL Server. See "Learning About Known Problems" on page 4-18 for more information about these lists.

Install or Upgrade SQL Server

Read through the entire SQL Server installation and configuration guide before you begin a new installation or upgrade. You need to plan parts of the installation and configure the operating system **before** installing SQL Server. It is also helpful to consult with the local administrator for your operating system to define operating system requirements for SQL Server. These requirements can include the configuration of memory, raw devices, asynchronous I/O, and other features, depending on the platform you use. Many of these tasks cannot be accomplished once the installation has started.

If you are upgrading a server, back up all data (including the *master* database, user databases, triggers, and system procedures) offline before you begin. After upgrading, immediately create a separate, full backup of your data, especially if there are incompatibilities between older dump files and the newer versions.

Install Additional Network Software

SQL Server generally includes support for the network protocol(s) that are common to your hardware platform. If your network supports additional protocols, install the required protocol support. As always, first read the *Release Bulletin* to make sure that the software is compatible with this SQL Server release.

Configure and Test Client Connections

A successful client connection depends on the coordination of SQL Server, the client software, and network products. If you are using one of the network protocols installed with SQL Server, see the SQL Server installation and configuration guide for information about testing network connections. If you are using a different network protocol, follow the instructions that are included with the network product. You can also use "ping" utilities that are included with Sybase connectivity products to test client connections with SQL Server.

Once you have successfully configured network connections, make a backup of the master interfaces file and use that file for all client connections. "The Interfaces File" on page 1-9 provides a general description of the interfaces file. See also the SQL Server installation and configuration guide for details about the name and contents of the file.

Allocating Physical Resources

Allocating physical resources is the process of giving SQL Server the memory, disk space, and CPU power required to achieve your performance and recovery goals. Every System Administrator must make decisions about resource utilization when installing a new server. You will also need to change SQL Server's resource allocation if you later upgrade your hardware platform by adding new memory, disk controllers, or CPUs or when the design of your database system changes. Or, early benchmarking of SQL Server and your applications can help you spot deficiencies in hardware resources that create performance bottlenecks.

See Chapter 5, “Overview of Disk Resource Issues,” in this manual to understand the kinds of disk resources that SQL Server requires. See also Chapter 8, “Configuring Memory,” and Chapter 10, “Managing Multiprocessor Servers,” for information about memory and CPU resources.

The following sections provide helpful pointers in determining physical resource requirements.

Dedicated vs. Shared Servers

The first step in the process of planning SQL Server resources is to understand the resources required by **other** applications running on the same machine. In most cases, System Administrators “dedicate” an entire machine for SQL Server use. This means that only the operating system and network software consume resources that otherwise might be reserved for SQL Server. On a “shared” system, other applications, such as SQL Server client programs or print servers, run on the same machine as SQL Server. It can be difficult to calculate the resources available to SQL Server on a shared system, because the types of programs and their pattern of use may change over time.

In either case, it is the System Administrator’s responsibility to take into account the resources used by operating systems, client programs, windowing systems, and so forth when configuring resources for SQL Server. Configure SQL Server to use only the resources that are available to it. Otherwise, the server may perform poorly or fail to start. Chapter 8, “Configuring Memory,” describes how to estimate the memory that SQL Server uses. See also Chapter 5, “Overview of Disk Resource Issues,” and Chapter 10, “Managing Multiprocessor Servers,” to understand how the server utilizes disk and CPU resources.

Decision Support and OLTP Applications

Determine, in advance, the required mix between the amount of online transaction processing (OLTP) work and decision support work that SQL Server is expected to perform. SQL Server contains many features that optimize performance for OLTP, decision support, and mixed workload environments. However, you must determine the requirements of your system’s applications in order to make optimal use of these features.

For mixed workload systems, list the individual tables that you anticipate will be most heavily used for each type of application. Maintaining a list of critical tables can help you when configuring named caches or partitioning data to achieve maximum performance for applications.

Plan Resource Usage in Advance

It is extremely important to understand and plan resource usage in advance. In the case of disk resources, for example, once you initialize and allocate a device to SQL Server, that device cannot be used for any other purpose (even if SQL Server never fills the device with data). Likewise, SQL Server automatically reserves the memory for which it is configured, and this memory cannot be used by any other application.

The following suggestions can help you plan resource usage:

- For recovery purposes, it is **always** best to place a database's transaction log on a separate physical device from its data. See Chapter 14, "Creating User Databases."
- Consider mirroring devices that store mission-critical data. See Chapter 7, "Mirroring Database Devices." You may also consider using disk arrays and disk mirroring for SQL Server data if your operating system supports these features.
- If you are working with a test SQL Server, you can initialize database devices as operating system files, rather than raw devices, for convenience. This helps you determine accurate sizes for the raw devices that you will create for the production server. See the SQL Server installation and configuration guide for information on configuring operating system files vs. raw devices.
- Keep in mind that changing configuration options can affect the way SQL Server consumes physical resources. This is especially true of memory resources. See Chapter 11, "Setting Configuration Parameters," for details about how much memory is used by individual parameters.

Configure the Operating System

Many times, you need to configure physical resources at the operating system level before you can make them available to SQL Server. After determining what resources are available to SQL Server and what resources you require, configure these resources at the operating system level:

- Initialize raw devices to the sizes required by SQL Server. If you initialize a raw device for SQL Server, that device cannot be used for any other purpose (for example, it cannot be used to store operating system files). Ask your UNIX administrator for assistance in initializing and configuring raw devices to the required sizes.
- Configure the number of network connections. The **number of user connections** parameter controls the maximum number of connections to SQL Server. However, you must make sure that the machine on which SQL Server runs can actually support that many connections. This is usually achieved by setting a variable in an operating system start-up file. See your operating system documentation.
- Additional configuration may be required for your operating system and the applications that you use. Read the SQL Server installation and configuration guide to understand the SQL Server operating system requirements. Also read your client software or consult with your engineers to understand the operating system requirements for your applications.

Backup and Recovery

Making regular backups of your databases is crucial to the integrity of your database system. Although SQL Server automatically recovers from system crashes (for example, power outages) or server crashes, only **you** can recover from data loss caused by media failure. Follow the basic guidelines below for backing up your system.

Chapters 17–20 in this manual describe how to develop and implement a backup and recovery plan.

Keep Up-to-Date Backups of Master

Backing up the *master* database is the cornerstone of any backup and recovery plan. The *master* database contains details about the structure of your entire database system. Its keeps track of the SQL Server databases, devices, and device fragments that make up those databases. Because SQL Server needs this information during recovery, it is crucial to maintain an up-to-date backup copy of the *master* database at all times.

To ensure that your backup of *master* is always up to date, back up the database after each command that affects disks, storage, databases, or segments. This means you should back up *master* after performing any of the following procedures:

- Creating or deleting databases
- Initializing new database devices
- Adding new dump devices
- Using any device mirroring command
- Creating or dropping system stored procedures, if they are stored in *master*
- Creating, dropping, or modifying a segment
- Adding new SQL Server logins

To back up *master* to a tape device, start **isql** and enter the command:

```
dump database master to "tape_device"
```

where *tape_device* is the name of the tape device (for example, */dev/rmt0*).

Keep Offline Copies of System Tables

In addition to backing up *master* regularly, keep offline copies of the contents of the following system tables: *sysdatabases*, *sysdevices*, *sysusages*, *sysloginroles*, and *syslogins*. Do this by using the **defncopy** utility, described in the SQL Server utility programs manual, and by storing a printed copy of the contents each system table. You can create a printed copy by printing the output of the following queries:

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

If you have copies of these tables, and a hard disk crash or some other disaster makes your database unusable, you will be able to use the recovery procedures described in Chapter 20, “Backing Up and Restoring the System Databases.”

You should also keep copes of all data definition language (DDL) scripts for user objects, as described under “Keeping Records” on page 3-11.

Automate Backup Procedures

Creating an automated backup procedure takes the guesswork out of performing backups and makes the procedure easier and quicker to perform. Automating backups can be as simple as using an operating system script or utility (for example, the UNIX **cron** utility) to perform the necessary backup commands. Or you can automate the procedure further using thresholds, which are discussed in Chapter 21, "Managing Free Space with Thresholds."

Although the commands required to create an automated script vary, depending on the operating system you use, all scripts should accomplish the same basic steps:

1. Start **isql** and dump the transaction log to a holding area (for example, a temporary file).
2. Rename the dump file to a name that contains the dump date, time, and database name.
3. Make a note about the new backup in a history file.
4. Record any errors that occurred during the dump in a separate error file.
5. Automatically send mail to the System Administrator for any error conditions.

A script that completes the above tasks simplifies the process of making incremental backups. Periodically, you should also verify the database's consistency using **dbcc**, dump the entire database to tape, and purge the incremental log files.

Verify Data Consistency Before Backing Up a Database

Having backups of a database sometimes is not enough—you must have consistent, **accurate** backups (especially for *master*). If you back up a database that contains internal errors, the database will have the same errors when you restore it.

Using the **dbcc** commands, you can check a database for errors before backing it up. **dbcc** commands verify that the storage of a database or database object "makes sense" to SQL Server. Always use **dbcc** commands to verify the integrity of a database before dumping it. If **dbcc** detects errors, correct them before dumping the database.

Over time, you can begin to think of running **dbcc** as insurance for your databases. If you discovered few or no errors while running **dbcc** in the past, you may decide that the risk of database corruption is small and that **dbcc** can be run only occasionally. Or, if the consequences of losing data are too high, you should continue to run **dbcc** commands each time you back up a database.

➤ ***Note***

For performance considerations, many sites choose to run **dbcc** checks outside of peak hours or on separate servers.

See Chapter 17, “Checking Database Consistency,” for information about the **dbcc** command.

Monitor the Log Size

When the transaction log becomes nearly full, it may be impossible to use standard procedures to dump transactions and reclaim space. The System Administrator should monitor the log size and perform regular transaction log dumps (in addition to regular database dumps) to make sure this situation never occurs. Use the preferred method of setting up a threshold stored procedure to automatically notify you (or dump the log automatically) when the log reaches a certain capacity. See Chapter 21, “Managing Free Space with Thresholds,” for information about using threshold procedures. It is also good to dump the transaction log just prior to doing a full database dump in order to shorten the time required to dump and load the database.

You can also monitor the space used in the log segment manually by using the **sp_helpsegment** stored procedure, as described under “Getting Information About Segments” on page 16-16.

Ongoing Maintenance and Troubleshooting

In addition to making regularly scheduled backups, the System Administrator performs the following maintenance activities throughout the life of a server.

Starting and Stopping SQL Server

Most system administrators automate the procedure for starting SQL Server to coincide with the start-up of the server machine. This can be accomplished by editing operating system start-up scripts or through other operating system procedures. See the SQL Server installation and configuration guide to determine how to start and stop SQL Server.

Viewing and Pruning the Error Log

The System Administrator should examine the contents of the error log on a regular basis to determine if any serious errors have occurred. You can also use operating system scripts to scan the error log for particular messages, and to notify the system administrator when specific errors occur. Checking the error log regularly helps you determine whether there are continuing problems of the same nature or if a particular database device is going bad. See Chapter 4, “Diagnosing System Problems,” for more information about error messages and their severity.

The error log file can grow large over time, since SQL Server appends informational and status messages to it each time it starts up. You can periodically “prune” the log file by opening the file and deleting old records. Keeping the log file to a manageable size saves disk space and makes it easier to locate current errors.

Keeping Records

Keeping records about your SQL Server system is an important part of your job as a System Administrator. Accurate records of changes and problems that you have encountered can be a valuable reference when contacting Technical Support or recovering databases. More important, they can provide vital information for administrators who manage the SQL Server system in your absence. The following sections outline the kinds of records that are most valuable to maintain.

Contact Information

Maintain a list of contact information for yourself (the System Administrator) as well as the System Security Officer, Operator, and database owners on your system. Also, record secondary contacts for each role. Make this information available to all SQL Server users so that the appropriate contacts receive enhancement requests and problem reports.

Configuration Information

Ideally, you should create databases, create database objects, and configure SQL Server using script files that you later store in a safe place. Storing the script files after use makes it possible to re-create your entire system in the event of a disaster. It also allows you to recreate database systems quickly on new hardware platforms for evaluation purposes. If you use a third-party tool to perform system administration, remember to generate equivalent scripts after performing administration tasks.

Consider recording the following kinds of information:

- Commands used to create databases and database objects (DDL scripts)
- Commands that add new SQL Server logins and database users
- The current SQL Server configuration file, as described in “Using sp_configure with a Configuration File” on page 11-7
- The names, locations, and sizes of all files and raw devices initialized as database devices

It is also helpful to maintain a dated log of all changes to the SQL Server configuration. Mark each change with a brief description of when and why you made the change, as well a summary of the end result.

Maintenance Schedules

Keep a calendar of events for regularly scheduled maintenance activities. Such a calendar should list any of the procedures you perform at your site:

- Using **dbcc** to check database consistency
- Backing up user and system databases
- Monitoring the space left in transaction logs (if this is not done automatically)
- Dumping the transaction log on a regular basis
- Examining the error log contents for SQL Server, Backup Server™, and SQL Monitor Server™
- Running the **update statistics** command (see “Index Statistics” on page 6-35 in the *Performance and Tuning Guide*)

- Examining auditing information, if you are installing the auditing option
- Recompiling stored procedures
- Monitoring the resource utilization of the server machine

System Information

Record information about the hardware and operating system on which you run SQL Server. This can include:

- Copies of operating system configuration files or start-up files
- Copies of network configuration files (for example, the *hosts* and *services* files)
- Names and permissions for the SQL Server executable files and database devices
- Names and locations of the tape devices used for backups
- Copies of operating system scripts or programs for automated backups, starting SQL Server, or performing other administration activities

Disaster Recovery Plan

Consolidate the basic backup and recovery procedures, the hints provided in “Backup and Recovery” on page 3-7, and your personal experiences in recovering data into a concise list of recovery steps tailored to your system. This can be useful both to yourself and to other System Administrators who may need to recover a production system in the event of an emergency.

Getting More Help

The amount of new information that System Administrators must learn can sometimes seem overwhelming. However, there are several software tools that can help you in learning and facilitating basic administration tasks. These include SQL Server Monitor, used for monitoring server performance and other activities, and SQL Server Manager™, which assists in many different administration tasks. Also available are many third-party software packages designed to help System Administrators manage daily maintenance activities.

4 Diagnosing System Problems

This chapter discusses diagnosing and fixing system problems. Topics covered include:

How SQL Server Responds to System Problems

When SQL Server encounters a problem—whether it is 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, this is what happens if you make a typing error and try to access a table that does not exist:

```
select * from publishers

Msg 208, Level 16, State 1:
Invalid object(table) name publishers
```

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 SQL Server with us_english as the default language):

```
select error, severity, description
from sysmessages
where error >=101 and error <=106
and langid is null
```

```
error severity description
----- -------- --------------------------------------------------
  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 SQL92.
- 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 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, and so forth), or the name of a particular database object.

In the *description* field of *sysmessages*, a percent sign (%) followed by a character or character string serves as a placeholder for these pieces of data, which SQL Server supplies when it encounters the problem and generates the error message. “%d” is a placeholder for a number; “%S_MSG” is a placeholder for a kind of database object; “%.*s”—all within quotes—is a placeholder for the name of a particular database object. Table 4-1 lists placeholders 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 “Reporting Errors” on page 4-11.)

Variables in Error Message Text

The following chart explains the abbreviations that appear in the error message text provided with each error message explanation:

Table 4-1: Error text abbreviation key

Symbol	Stands For
%d, %D	Decimal number
%x, %X, %.*x, %lx, %04x, %08lx	Hexadecimal number
%s	Null-terminated string

Table 4-1: Error text abbreviation key (continued)

Symbol	Stands For
%.*s, %*s, %*.s	String, usually the name of a particular database object
%S_*type*	SQL Server-defined structure
%c	Single character
%f	Floating-point number
%ld	Long decimal
%lf	Double floating-point number

Error Logging

Most error messages from SQL Server are sent only to the user's screen.

The backtrace from fatal error messages (severity levels 19 and higher) and error messages from the kernel are also sent to an error log file. The name of this file varies; see the SQL Server installation and configuration guide or the SQL Server utility programs manual.

➤ ***Note***

The error log file is owned by the user who installed SQL Server (or the person who started SQL Server after an error log was removed). Permissions or ownership problems with the error log at the operating system level can block successful start-up of SQL Server.

SQL Server creates an error log for you if one 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 an option 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 option that specifies the location of the errorlog (**-e** on UNIX and PC platforms; **/errorfile** on OpenVMS).
- If you do not include the error log option, the error log is created in the directory where you start SQL Server.

➤ ***Note***

Always start 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 server, 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 by deleting 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 2-digit number followed by a colon at the beginning of each line.
- The date, displayed in the 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.

Following is an example of a line from an error log:

```
00: 95/02/18 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, 1995, 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. A few corresponding messages differ in severity levels, so you may occasionally 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 (for example, 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 the *SQL Server Reference Manual* for more information.

In **isql** and Data Workbench®, severity level 10 (status or informational) messages do not display a message number or severity level. In Open Client™ applications, SQL Server returns "0" for severity level 10.

Severity levels 1–16 indicate problems that are caused by mistakes in what users have entered.

Severity levels 16 and higher indicate software or hardware errors. If the number is 17 or 18, you can continue with the work you are doing, although you may not be able to execute a particular command.

Severity levels 19 and higher indicate 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.

Fatal errors (levels 19 and higher) 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 do not necessarily damage a database or its objects, but they can. They may also result from earlier damage to a database or its objects. 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 higher 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 17–24.

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 have to be reloaded.

For more information, refer to the following:

- **dbcc** is discussed in Chapter 17, "Checking Database Consistency."
- Loading a user database is discussed in Chapter 19, "Backing Up and Restoring User Databases."
- Loading system databases is discussed in Chapter 20, "Backing Up and Restoring the System Databases."

The following subsections discuss each severity level.

Levels 10–18

Error messages with severity levels 10–16 are generated by problems that are caused by user errors. These problems 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 and 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 been run, SQL Server displays a message telling the user how much of the requested space has been allocated for the new database.

Level 11: Specified Database Object Not Found

Messages with severity level 11 indicate that SQL Server cannot find an object that was 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 the spelling of the object names, use the owner names if the object is not owned by you or “dbo”, and make sure you are 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 in 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 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 back a transaction 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 that you do not 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 Statement

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 nonfatal mistake that does not 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 the command syntax and working database context.

➤ ***Note***

The System Administrator should monitor occurrences of errors with severity levels 17–24. Levels 17 and 18 are usually not reported in the error log. Users should be instructed to notify the System Administrator when level 17 and 18 errors 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 11, “Setting Configuration Parameters,” for more information on changing configuration variables.

The Database Owner can correct the 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 do not keep users from their work, users tend 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 19–24

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 a command. The problem has affected only the current process, and 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 was previously damaged by a software or hardware problem.

The first step is to restart SQL Server and run **dbcc** to determine whether other objects in the database are also damaged. Whatever the report from **dbcc** may be, it is 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 does not help, then 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: Fatal Error: Database Integrity Suspect

Error messages with severity level 23 mean that the integrity of the entire database is suspect due to previous damage caused by a software or hardware problem. Restart SQL Server and run the **dbcc** diagnostics.

Even when a level 23 error indicates that the entire 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, that is, 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 one 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 an option that specifies the location of the error log.
- If you type the **backupserver** command from the operating system, you can include the option that specifies the location of the error log (**-e** on UNIX and PC platforms; **/errorfile** on OpenVMS).
- If you do not include the error log option, the error log is created in the directory where you start Backup Server.

Backup Server error messages 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 in 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. The 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 that is 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 is not 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:

Table 4-2: Status values reported by sp_who

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.
`running`	Actively running on one of 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": when space is freed in the log by a **dump transaction** command or when an SA uses the **lct_admin** function to wake up "log suspend" processes.

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
```

In the example above, processes 2–5 cannot be killed: they are system processes. The login name NULL and the lack of a host name 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 and 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**, 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 has been blocked while 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 the *Performance and Tuning Guide*.

Shutting Down Servers

A System Administrator can shut down SQL Server or 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:

1. Disables logins, except for System Administrators
2. Performs a checkpoint in each database, flushing pages that have changed from memory to disk
3. Waits for currently executing SQL statements or procedures to finish

In this way **shutdown** 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 the Backup Servers that are 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 reissue the full set of load commands, starting with **load database.**

Learning About Known Problems

The *Release Bulletin* is a valuable resource for learning about known problems or incompatibilities with SQL Server and Backup Server. Reading the *Release Bulletin* in advance can the time and guesswork of troubleshooting known problems.

The SQL Server installation program also installs files that list all System Problem Reports (SPRs) and Closed Problem Reports (CPRs) for SQL Server release 11.0. Problem reports are organized by functional areas of the product. For example, a file named *cpr_bus* would contain a listing of closed (fixed) problem reports pertaining to the Backup Server, and the file *spr_bus* would contain a list of currently open problem reports for the Backup Server.

See the *Release Bulletin* to learn the location of CPR and SPR files.

Managing Physical Resources

5 Overview of Disk Resource Issues

This chapter discusses some basic issues that determine how you allocate and use disk resources with SQL Server. It covers the following topics:

SQL Server can make reasonable default decisions about many aspects of storage management, such as where databases, tables, and indexes are placed and how much space is allocated for each one. However, the System Administrator has ultimate control over the allocation of disk resources to SQL Server and the physical placement of databases, tables, and indexes on those resources.

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

Device Allocation and Object Placement

When configuring a new system, the System Administrator must consider several issues that have a direct impact on the number and size of disk resources required. These device allocation issues refer to commands and procedures that add disk resources to SQL Server. Device allocation topics are described in chapters shown in Table 5-1.

Table 5-1: Device allocation topics

Task	Chapter
Initialize and allocate a default pool of database devices.	Chapter 6, “Initializing Database Devices”
Mirror database devices for recovery.	Chapter 7, “Mirroring Database Devices”

After the initial disk resources have been allocated to SQL Server, the System Administrator, Database Owner, and object owners should consider how to place databases and database objects on specific database devices. These object placement issues determine where database objects reside on your system and whether or not the objects share devices. Object placement tasks are discussed throughout this manual, including the chapters shown in Table 5-2.

Table 5-2: Object placement topics

Task	Chapter
Place databases on specific database devices.	Chapter 14, “Creating User Databases”
Place tables and indexes on specific database devices.	Chapter 16, “Creating and Using Segments”

The concept of allocating devices should not be considered separately from the concept of object placement. For example, if you decide that a particular table must reside on a dedicated pair of devices, you must first allocate those devices to SQL Server. The remaining sections in this chapter provide an overview that spans both device allocation and object placement issues, providing pointers to chapters where appropriate.

Commands for Managing Disk Resources

Table 5-3 illustrates the major commands a System Administrator uses to allocate disk resources to SQL Server and provides references to the chapters that discuss those commands.

Table 5-3: Commands for allocating disk resources

Command	Task	Chapter
disk init **name = "*dev_name*"** **physname = "*phys_name*"...**	Makes a physical device available to a particular SQL Server. Assigns a database device name (*dev_name*) that is used to identify the device in other SQL Server commands.	Chapter 6, “Initializing Database Devices”
sp_diskdefault "*dev_name*"...	Adds *dev_name* to the general pool of default database space.	Chapter 6, “Initializing Database Devices”
disk mirror **name = *"dev_name"*** **mirror = "*phys_name"*...**	Mirrors a database device on a specific physical device.	Chapter 7, “Mirroring Database Devices”

The System Administrator, Database Owner, and object owners must decide how to place databases and database objects on specific devices. Table 5-4 illustrates the commands used in object placement. See also Chapter 13, "Controlling Physical Data Placement,"in the *Performance and Tuning Guide* for information about how object placement affects performance.

Table 5-4: Commands for placing objects on disk resources

Command	Task	Chapter
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.	Chapter 14, "Creating User Databases"
create database... or **alter database...**	When used without the **on** ***dev_name*** clause, these commands allocate space on the default database devices.	Chapter 14, "Creating User Databases"
sp_addsegment ***seg_name*****,** ***dbname*****,** ***devname*** and **sp_extendsegment** ***seg_name*****,** ***dbname*****,** ***devname***	Creates a segment, a named collection of space, from the devices available to a particular database.	Chapter 16, "Creating and Using Segments"
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.	Chapter 16, "Creating and Using Segments"
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).	Chapter 16, "Creating and Using Segments"

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 placing database objects

to physical devices yields performance improvements. Disk mirroring slows the speed of disk writes.

Recovery

Recovery is the key motivation for using several disk devices. Nonstop 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 re-created by loading a dump of the database and then applying the log records that were safely stored on another device. See Chapter 14, "Creating User Databases," for information about the **log on** clause of **create database**.

Mirroring

Nonstop 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. Chapter 7, "Mirroring Database Devices," describes the process of mirroring devices.

Performance

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

- Placing a table on one hard disk and nonclustered 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.

See Chapter 13, “Controlling Physical Data Placement,” in the *Performance and Tuning Guide* for a detailed discussion of how object placement affects performance.

Status and Defaults at Installation Time

Instructions for installing SQL Server are provided in the SQL Server installation and configuration 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 system databases, system defined segments, and database devices are organized as follows:

- 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*.

➤ ***Note***

After initializing new devices for default storage, remove the master device from the default storage area with **sp_diskdefault**. Do not store user databases and objects on *d_master*. See “Designating Default Devices” on page 6-7 for more information.

- If you have chosen to install the audit database, *sybsecurity*, it is located on its own device.

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 5-1.

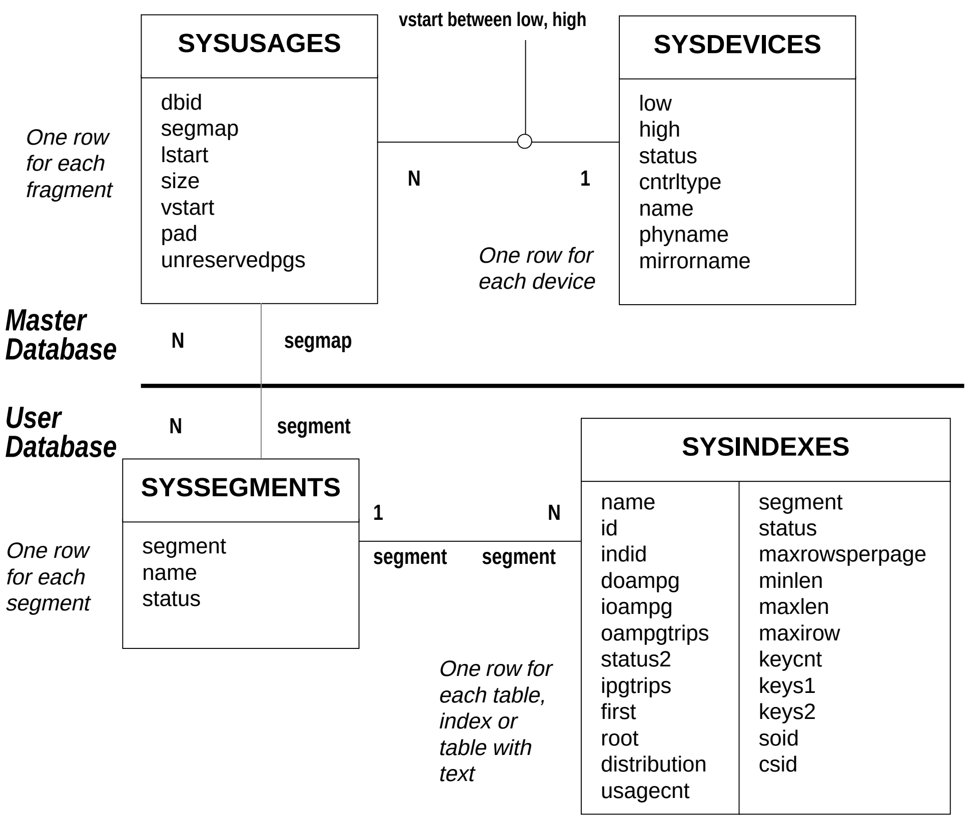


Figure 5-1: The system tables that manage storage

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 **sp_addumpdevice**, are discussed in Chapter 18, “Developing a Backup and Recovery Plan.”

sysdevices stores two names for each device:

- A **logical name** or **device name**, used in all subsequent storage-management commands, is 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** is the actual operating system name of the device. You use this name only 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 the **create database** or **alter database** statement. The **log on** clause to **create database** places a database's transaction log on a separate device 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. See Chapter 14, "Creating User Databases," for information about creating databases on specific database devices.

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. Chapter 16, "Creating and Using Segments," discusses these procedures in detail.

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 therefore 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 chain of text pages is stored. It also lists other information such as the **max_rows_per_page** setting for the table or index.

The **create table**, **create index**, and **alter table** commands create new rows in *sysindexes*. Partitioning a table changes the function of *sysindexes* entries for the table, as described under “Partitioning and Unpartitioning Tables” on page 13-15 of the *Performance and Tuning Guide*.

6 Initializing Database Devices

This chapter explains how to initialize database devices and how to assign devices to the default pool of devices. It includes the following sections:

Introduction

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 a disk (such as a disk partition) or a file in the file system that is 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 has been initialized, it can be:

- Allocated to the default pool of devices for the **create** and **alter database** commands
- Assigned to the pool of space available to a user database
- Assigned to a user database and used to store one or more database objects
- Assigned to store a database’s transaction logs

Using the *disk init* Command

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 SQL Server installation and configuration guide for your platform for information about choosing a database device and preparing it for use with SQL Server. You may want to repartition 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 1/2MB. 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 20, “Backing Up and Restoring the System Databases.”

disk init Syntax

The syntax of the **disk init** command is:

```
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 file names for *physicalname*.

Choosing a Device Number for *disk init*

vdevno is an identifying number for the database device. It must be unique among the 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 1–9. To see the configuration value for your system, execute **sp_configure "number of devices"**, without giving a second parameter, and check the run value, the last column in the output:

```
              sp_configure "number of devices"
Parameter name  Default  Memory Used  Config Value  Run Value
-------------- -------  -----------  ------------  ----------
number of devices     10            0            10         10
```

To see the numbers already in use for **vdevno**, look in the *device_number* column of the report from **sp_helpdevice**, or use the following query to list 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 parameters, in Chapter 11, “Setting Configuration Parameters.”

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

Specifying the Device Size with *disk init*

The **size** of the database device must be given in 2K blocks. There are 512 2K blocks in 1MB. The maximum size of a database device is 1,048,576 2K blocks (2GB).

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

- The size of *model*. When you install SQL Server, *model* uses 1024 2K blocks (2MB). Use **sp_helpdb model** to see the current size of *model*.
- The configuration parameter **default database size**. Use **sp_configure** and look at the run value for **default 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 (1MB).

If you are initializing a raw device (UNIX) or a foreign device (OpenVMS), determine the size of the device from your operating system, as described in the SQL Server installation and configuration guide for your platform. Use the total size available, up to the maximum of 2GB. Once you have initialized the disk for use by SQL Server, you cannot use any space on the disk for any other purpose.

For this reason, you should never create a raw device or foreign device larger than 2GB if you plan to use it as a SQL Server database device.

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

◆ ***WARNING!***

If the physical device does not contain the number of blocks specified by the size parameter, the disk init command fails. If you use the optional vstart parameter, the physical device must contain the sum of the blocks specified by both the vstart and size parameters, or the command fails.

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 0. If the specified devices does not have the sum of **vstart** + **size** blocks available, the disk init command fails.

The optional **cntrltype** keyword specifies the disk controller. Its default value is 0. 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.

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 about 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 that 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. The status bits and their meanings are listed in the following table:

Table 6-1: Status bits in sysdevices

Bit	Meaning
1	Default disk (may be used by any **create** or **alter database** command that does not specify a location)
2	Physical disk
4	Logical disk (not used)
8	Skip header (used with tape dump devices)
16	Dump device
32	Serial writes

Table 6-1: Status bits in sysdevices (continued)

Bit	Meaning
64	Device mirrored
128	Reads mirrored
256	Secondary mirror side only
512	Mirror enabled
2048	Used internally; set after **disk unmirror, side = retain**

For more information about dump devices and **sp_addumpdevice**, see Chapter 18, "Developing a Backup and Recovery Plan."

Dropping Devices

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

```
sp_dropdevice logicalname
```

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. See the SQL Server installation and configuration guide for information about starting and stopping SQL Server.

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**. The **sp_diskdefault** command 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 logicalname, {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
```

SQL Server can have multiple default devices. They are used in the order in which they appear in the *sysdevices* table (that is, 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 are 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.

Placing Objects on Database Devices

After initializing a set of database devices, you may want to assign them to specific databases or database objects rather than adding them to the default pool of devices. For example, you may want to make sure a table never grows beyond the size of a particular device.

Refer to these chapters for information about placing databases and objects on devices:

- Chapter 14, “Creating User Databases,” describes how to place user databases on specific database devices.
- Chapter 16, “Creating and Using Segments,” explains how to use segments to place tables and indexes on specific devices.

7 Mirroring Database Devices

This chapter describes the process of creating and administering disk mirrors. It includes the following sections:

Introduction

Disk mirroring can provide nonstop recovery in the event of media failure. The **disk mirror** command causes a SQL Server database device to be duplicated, that is, 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.

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

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.

➤ ***Note***

You cannot mirror a dump device.

Figure 7-1 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 disk 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.

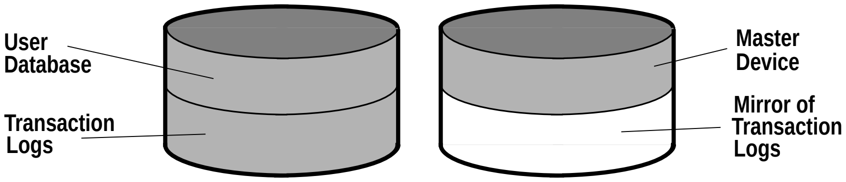


Figure 7-1: Disk mirroring using minimal physical disk space

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 nonstop recovery because the *master* and user databases are not being mirrored, and must be recovered from backups.

Figure 7-2 represents another mirror configuration. In this case, the master device, user databases, and 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 7-2 provides nonstop 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.

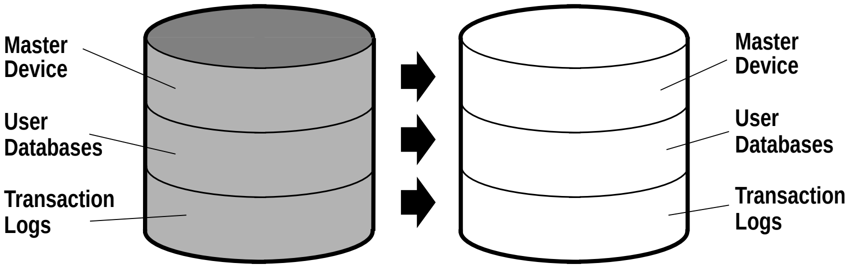


Figure 7-2: Disk mirroring for rapid recovery

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

Figure 7-3 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.

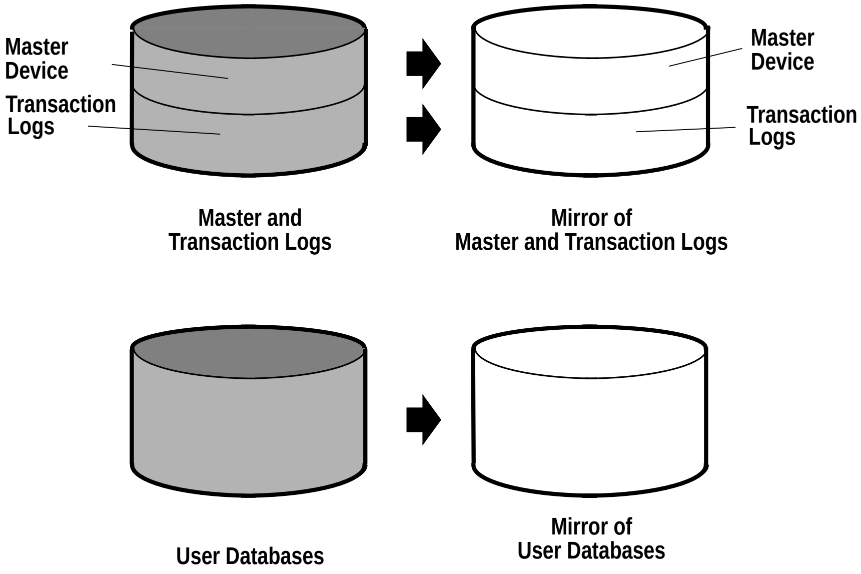


Figure 7-3: Disk mirroring: keeping transaction logs on a separate disk

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

1. **Speed of recovery**. You can achieve nonstop 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 7-2 and Figure 7-3) increases the time needed to write transactions to both disks.

Conditions That Do Not Disable Mirroring

SQL Server disables a mirror only when it encounters an I/O error on a mirrored device. For example, if SQL Server tries to write to a bad block on the disk, the resulting error disables mirroring for the device. However, processing continues without interruption on the unaffected mirror.

The following conditions **do not** disable a mirror:

- An unused block on a device is bad. SQL Server does not detect an I/O error and disable mirroring until it accesses the bad block.
- Data on a device is overwritten. This might happen if a mirrored device is mounted as a UNIX file system, and UNIX overwrites the SQL Server data. This causes database corruption, but mirroring is not disabled, since SQL Server would not encounter an I/O error.
- Incorrect data is written to both the primary and secondary devices.
- The file permissions on an active device are changed. Some System Administrators may try to test disk mirroring by changing permissions on one device, hoping to trigger I/O failure and unmirror the other device. But the UNIX operating system does not check permissions on a device after opening it, so the I/O failure does not occur until the next time the device is started.

Disk mirroring is not designed to detect or prevent database corruption. Some of the scenarios described can cause corruption, so you should regularly run consistency checks such as **dbcc checkalloc** and **dbcc checkdb** on all databases. See Chapter 17, "Checking Database Consistency," for a discussion of these commands.

Disk Mirroring Commands

The **disk mirror**, **disk unmirror**, and **disk remirror** commands control disk mirroring. All 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***

The **disk mirror**, **disk unmirror**, and **disk remirror** 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, always mirror devices that are capable of asynchronous I/O to other devices capable of asynchronous I/O. In most cases, this means mirroring raw devices to raw devices and operating system files to operating system files.

If the operating system cannot perform asynchronous I/O on files, mirroring a raw device to a regular file produces an error message. Mirroring a regular file to a raw device will work, but will not use asynchronous I/O.

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 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, that is, 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 Reference 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 have chosen. Table 6-1 on page 6-6 describes the status bits.

Unmirroring a Device

Disk mirroring is automatically deactivated 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 (see Table 6-1 on page 6-6). Its effects on the *phyname* and *mirrorname* columns in *sysdevices* depend on the **side** argument also, as shown in Table 7-1.

Table 7-1: Effects of mode and side options to the disk mirror command

		side	
		primary	**secondary**
mode	**remove**	Name in *mirrorname* moved to *phyname* and *mirrorname* set to null; *status* changed	Name in *mirrorname* removed; *status* changed
	retain	Names unchanged; *status* changed to indicate which device is being deactivated	

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 want to add certain warnings in applications that perform updates, or use **sp_dboption** to make certain databases read only if the disk becomes unmirrored.

➤ ***Note***

SQL Server knows that a device has become unmirrored only when it attempts I/O to the mirror device. On mirrored databases, this occurs at a checkpoint or when the SQL Server buffer must be written to disk. On mirrored logs, I/O occurs when 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 error log on mirror failure) are activated only 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 SQL Server installation and configuration guide.

Disk Mirroring Tutorial

The following steps illustrate the use of disk mirroring commands and their effect on selected columns of *master..sysdevices*.

Step 1

Initialize a new test device using the command:

```
disk init name = "test",
physname = "/usr/sybase/test.dat",
size=5120, vdevno=3
```

This inserts the following values into columns of *master..sysdevices*:

```
name  phyname               mirrorname   status
test  /usr/sybase/test.dat   NULL         2
```

Status 2 indicates that the device is a physical disk. Since the device mirrored bit (64) is off and the *mirrorname* column is null, this device is not mirrored.

Step 2

Mirror the test device using the command:

```
disk mirror name = "test",
mirror = "/usr/sybase/test.mir"
```

This changes the *master..sysdevices* columns to:

```
name  phyname               mirrorname            status
test  /usr/sybase/test.dat  /usr/sybase/test.mir  738
```

Status 738 indicates that mirroring is currently active (512) on this device. Reads are mirrored (128), and writes are mirrored (64) and serial (32). The device is a physical disk (2).

Step 3

Disable the mirror device (the secondary side), but retain that mirror:

```
disk unmirror name = "test",
side = secondary, mode = retain
```

```
name  phyname               mirrorname            status
test  /usr/sybase/test.dat  /usr/sybase/test.mir  2274
```

Status 2274 indicates that mirror device has been retained (2048) but mirroring has been disabled (512 bit off), and only the primary device is used (256 bit off). Reads are mirrored (128), and writes are mirrored (64) and serial (32). The device is a physical disk (2).

Step 4

Remirror the test device:

```
disk remirror name = "test"
```

This resets the *master..sysdevices* columns to:

```
name  phyname               mirrorname            status
test  /usr/sybase/test.dat  /usr/sybase/test.mir  738
```

Status 738 indicates that mirroring is currently active (512) on this device. Reads are mirrored (128), and writes are mirrored (64) and serial (32). The device is a physical disk (2).

Step 5

Disable the test device (the primary side), but retain that mirror:

```
disk unmirror name = "test",
side = "primary", mode = retain
```

This changes the *master..sysdevices* columns to:

```
name  phyname               mirrorname            status
test  /usr/sybase/test.dat  /usr/sybase/test.mir  482
```

Status 482 indicates that mirroring has been disabled (512 bit off) and that only the secondary device is used (256). Reads are mirrored (128), and writes are mirrored (64) and serial (32). The device is a physical disk (2).

Step 6

Remirror the test device:

```
disk remirror name = "test"
```

This resets the *master..sysdevices* columns to:

```
name  phyname               mirrorname             status
test  /usr/sybase/test.dat  /usr/sybase/test.mir   738
```

Status 738 indicates that mirroring is currently active (512) on this device. Reads are mirrored (128), and writes are mirrored (64) and serial (32). The device is a physical disk (2).

Step 7

Disable the test device (the primary side), and remove that mirror:

```
disk unmirror name = "test", side = "primary",
mode = remove
```

This changes the *master..sysdevices* columns to:

```
name  phyname               mirrorname             status
test  /usr/sybase/test.mir  NULL                   2
```

Status 2 indicates that the device is a physical device. Since the *mirrorname* column is null, mirroring is not enabled on this device.

Step 8

Remove the test device to complete the tutorial:

```
sp_dropdevice test
```

This removes all entries for the test device from *master..sysdevices*.

8 Configuring Memory

This chapter describes how SQL Server uses memory and explains how to maximize the memory available to SQL Server on your system. It also explains how SQL Server divides the user-configurable portion of SQL Server memory into the procedure cache and data cache. This chapter contains the following sections:

After following the instructions in this chapter, refer to Chapter 9, “Configuring Data Caches,” to partition the data cache into named caches and bind objects to caches.

Maximizing SQL Server Memory

The more memory that is available, the more resources SQL Server has for internal buffers and caches. Having enough memory available for caches reduces the number of times SQL 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, be sure to assess other memory needs on your system, and make sure that SQL Server uses only the remaining available memory. SQL Server may not be able to boot if it cannot acquire the memory for which it is configured.

To determine the maximum amount of memory available for SQL Server on your system:

1. Determine the total amount of physical memory on your computer system.
2. Subtract the memory required for the operating system from the total physical memory.
3. If the machine is not dedicated to SQL Server, subtract the memory requirements for other system uses also. For example, subtract the memory that will be used by any client applications that will run on the SQL Server machine. 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.

4. Subtract any memory that you want to allocate for the **additional network memory** configuration parameter. This is explained in "additional network memory" on page 11-62.

The memory left over after subtracting requirements for the operating system, other applications, and **additional network memory** is the total memory available for SQL Server. Configure SQL Server to use this remaining memory by setting the **total memory** parameter to that value. See "total memory" on page 11-64 for details on setting **total memory** and other configuration parameters.

Consider changing the value of the **total memory** configuration parameter:

- When you change the amount of RAM on your machine
- When the pattern of use of your machine changes
- If you allocate memory for **additional network memory** for SQL Server

If SQL Server Cannot Boot

When SQL Server boots, 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. If the configured size is larger than the available memory, SQL Server cannot boot.

➤ ***Note***

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.

If SQL Server does not boot for this reason, reduce the memory requirements for SQL Server by editing the value of the **total memory** parameter (or other parameters that use significant amounts memory) in the server's configuration file. Then reboot SQL Server to use the new values. See Chapter 11, "Setting Configuration Parameters," for information about using configuration files.

How SQL Server Uses Memory

The run value of the **total memory** parameter specifies the total amount of memory that SQL Server requires at start-up. For example, if the **total memory** parameter has a value of 10,000 pages, SQL Server tries to obtain 19.5MB (10,000 * 2048) of memory at start-up. If this amount is not available, SQL Server will not start.

When SQL Server starts, it allocates memory for:

- SQL Server executable code
- Static memory used by SQL Server
- Memory for user-configurable parameters
- Non-cache data structures

The remaining memory is then divided between the SQL Server caches:

- Data cache
- Procedure cache (based on the value of the **procedure cache percent** parameter)

The size of the data and procedure caches has a significant impact on overall performance. On a development system, you may want to increase the amount of memory dedicated to the procedure cache. On a production system, however, you may want to reduce the size of the procedure cache again in order to obtain more memory for the data cache. See Chapter 15, “Memory Use and Performance,” in the *Performance and Tuning Guide* for recommendations on optimizing procedure cache size.

➤ ***Note***

SQL Server release 11.0 requires additional memory for scratch space when compiling stored procedures. Although this memory is freed after compilation, you may need to set the **procedure cache percent** configuration parameter higher than for previous SQL Server releases.

To determine the amount of memory available for caches, you can estimate the amount of overhead required for SQL Server and subtract that amount from the **total memory.** Or you can directly compute the size of the caches using start-up messages from the error log file.

Estimating SQL Server Overhead

This section explains how to estimate the total cache size by subtracting SQL Server overhead from **total memory.** It uses an example **total memory** configuration of 7500 pages, or 14.65MB:

total memory= (7500 pages) * (1MB / 512 pages) = 14.65MB

Figure 8-1 shows how memory is allocated for SQL Server:

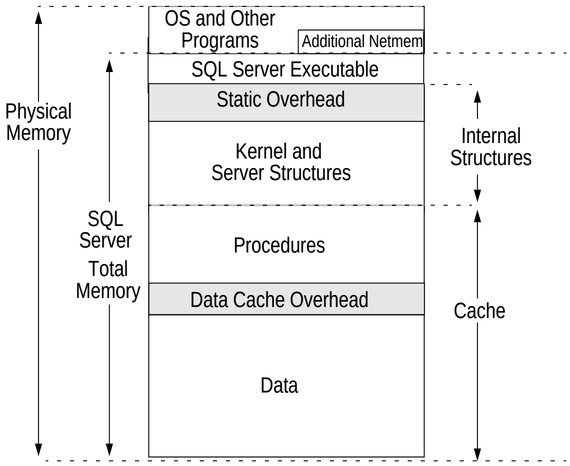


Figure 8-1: Memory allocation example

SQL Server Executable Code

The size of the SQL Server executable code must be subtracted from the total memory available to the SQL Server process. The size of the executable code varies by platform and release, but generally ranges from 3MB to 4MB. To determine the size of the SQL Server executable for your platform, use **sp_configure** to display the value of the **executable code size** parameter. See “executable code size” on page 11-47 for more information.

This example uses a code size of 3.26MB. Subtracting this amount from **total memory** leaves 11.39MB.

Internal Structures

After the size of the executable has been subtracted, an additional amount of memory is allocated for SQL Server internal structures. Internal structures consist of both kernel and server structures, but it is easiest to think of this memory as a combination of static overhead and memory for user-configurable parameters.

Static Overhead

SQL Server allocates a certain amount of memory as static overhead. This amount is not affected by user-configurable parameters and normally varies between 2.2MB and 3.25MB. Subtract static memory from total memory when estimating the memory available for caches on your system.

In the current example, SQL Server uses a value of 3MB for static overhead. Subtracting static overhead from the remaining memory leaves 8.39MB.

Memory for User-Configurable Parameters

SQL Server also allocates memory for the user-configurable parameters. The total amount of memory required depends on both the type of configuration parameter and its assigned value. Chapter 11, “Setting Configuration Parameters,” contains a detailed description of all configuration parameters and the memory each requires.

Table 8-1 contains a list of parameters to consider for estimating the memory that SQL Server uses. This list includes parameters that use significant amounts of memory, but not parameters that are related to remote procedure calls. See the associated page number for more information about the parameter’s memory usage:

Table 8-1: Parameters for estimating SQL Server memory

Parameter	See Page
number of user connections	11-101
number of open databases	11-87
number of devices	11-28
number of open objects	11-88
number of locks	11-41

User-configurable parameters that are not listed in Table 8-1 generally require smaller amounts of memory. To get the most accurate estimate of memory used by parameters, use **sp_configure** with no other options to display the full list of parameters with the amount of memory each uses. Add the values in the "Memory Used" column of the **sp_configure** output to obtain the total memory used by configuration parameters. See Chapter 11, "Setting Configuration Parameters," for information about using **sp_configure.**

This example uses a total value of 3MB for user-configurable parameters. Subtracting 3MB of configurable overhead from the remaining memory of 8.39MB leaves 5.39MB for the total cache space.

Data and Procedure Caches

The proportion of the remaining memory (5.39MB in the example below) that goes to procedure cache depends on the run value of the **procedure cache percent** configuration parameter. A value of 20 indicates that 20 percent of the total cache space is used for procedure cache and the remaining 80 percent is used for data cache.

Therefore, with a **procedure cache percent** value of 20, this estimation would result in the following values for the data and procedure caches:

- Data cache = (5.39) * (0.8) = 4.31MB (or 2207 pages)
- Procedure cache = (5.39) * (0.2) = 1.08MB (or 552 pages)

Normally, the amount of procedure cache is slightly higher than the amount indicated in this calculation because a portion of the 6 percent of miscellaneous overhead that is not used is added to the procedure cache.

Effects of Increasing Memory

The amount of SQL Server overhead does not depend on the amount of memory available to it. You can add memory to SQL Server by increasing the value of the **total memory** parameter. This directly increases the amount of total cache space by the amount of memory added. Make sure that you do not increase SQL Server memory beyond available physical memory or the server will start to page fault.

For example, if you increase the **total memory** parameter from 7500 pages to 9548 pages in "Memory allocation example" on page 8-4, the data and procedure caches increase by 2048 pages (4MB).

Determining Total Cache Space from SQL Server Error Log

Another way to determine how SQL Server uses memory is to examine the memory-related messages written to the SQL Server error log when SQL Server starts. These messages state exactly how much data and procedure cache is allocated, as well as how many procedures or other compiled objects can reside in cache at any one time.

These messages provide the most accurate information regarding cache usage on SQL Server. As discussed earlier, the amount of memory allocated to data and procedure caches depends on the run value of the **procedure cache percent** configuration parameter.

The memory-related messages from an example SQL Server with a **total memory** parameter of 7500 are:

```
server: Number of proc buffers allocated: 556
server: Number of blocks left for proc headers: 629.
server: Memory allocated for the default data cache: 4144 Kb
```

Each of these error log messages is described below.

Procedure Cache Messages

```
server: Number of proc buffers allocated: 556
```

This message states the total number of proc buffers allocated in the procedure cache.

```
server: Number of blocks left for proc headers:629.
```

This message indicates the total number of proc headers available for use in the procedure cache.

proc buffer

A **proc buffer** (procedure buffer) is a data structure used to manage compiled objects (any stored procedure, trigger, rule, default, check constraint, or view) in the procedure cache. One proc buffer is used for every copy of a named object stored in the procedure cache. When SQL Server starts, it determines the number of proc buffers required and multiplies that value by the size of a single proc buffer

(76 bytes) to obtain the total amount of memory required. It then allocates that amount of memory, treated as an array of proc buffers. Unlike some data structures, proc buffers can span pages.

proc header

A **proc header** (procedure header) is where a compiled object (such as a stored procedure) is stored. Depending on the size of the object to be stored, one or more proc headers may be required. The total number of compiled objects that can be stored in the procedure cache is limited by the number of available proc headers or proc buffers, whichever is less. In this example, no more than 629 compiled objects are allowed. Because stored procedures often use up more than one page, the practical value of this number may be even lower.

The total size of procedure cache is the combined total of memory allocated to proc buffers (rounded up to the nearest page boundary) plus the memory allocated to proc headers. In this example, there are 556 proc buffers allocated, thus 21 pages are used for proc buffers:

(556 buffers) * (76 bytes / buffer) / (2048 bytes / page) = 21 pages

Added to the 629 pages allocated for proc headers, the total amount of memory reserved for procedure cache is 650 2K pages (1.27MB).

Data Cache Messages

```
server: Memory allocated for the default data cache: 4144 Kb
```

This message indicates how many page buffers are allocated to the default data cache. In this example, 2072 pages (4.05MB) are dedicated to the default cache.

If you configure named caches, the error log will contain additional messages for each cache. To determine the total cache size, you can add the sizes allocated to all named cache in the system. Or, use the **sp_helpcache** procedure to get detailed information about individual caches, objects bound to those caches, and the overhead associated with different cache sizes. See “Getting Information About Cache Bindings” on page 9-15 for more information.

9 Configuring Data Caches

Introduction

This chapter describes how to create and administer named caches on SQL Server. It includes the following sections:

The most common reason for administering data caches is to reconfigure them for performance. While this chapter is primarily concerned with the *mechanics* aspect of working with data caches, Chapter 15, “Memory Use and Performance,” in the *Performance and Tuning Guide* discusses performance concepts associated with data caches.

The Data Cache on SQL Server

The data cache holds the data, index, and log pages that are currently in use by SQL Server, as well as pages that SQL Server used recently. When you first install SQL Server, it has a single default data cache that is used for all data, index, and log activity. You can divide this cache by creating named data caches. Also, you can create pools within the named caches and the default cache to perform large I/Os. You can then bind a database, table (including the *syslogs* table), index, or text or image page chain to a named data cache.

Large I/O sizes enable SQL Server to perform data prefetching when the query optimizer determines that prefetching would improve performance. For example, an I/O size of 16K means that SQL Server can read an entire extent, or eight 2K pages, all at once rather than performing eight separate I/Os. See Chapter 7, “The SQL Server Query Optimizer,” in the *Performance and Tuning Guide* for details about the optimizer.

The process of configuring named data caches divides the default cache into separate cache structures. The named data caches that you create can be used only by databases or database objects that are explicitly bound to them. All objects not explicitly bound to named data caches use the default data cache.

SQL Server provides user-configurable data caches in order to improve performance, especially for multi-processor servers. A full discussion of the ways that configuring named data caches can improve performance appears in “Named Data Caches and Performance” on page 15-12 of the *Performance and Tuning Guide*.

Figure 9-1 shows a data cache with the default cache and two named data caches. The default cache contains two pools, a 2K pool and a 16K pool. The *User_Table_Cache* cache has a 2K pool and a 16K pool. The *Log_Cache* has a 2K pool and a 4K pool.

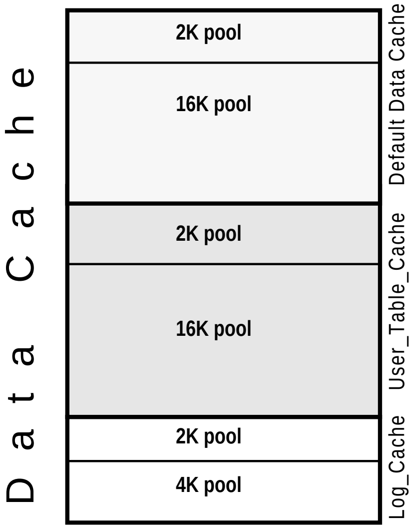


Figure 9-1: Data cache with default cache and two named data caches

Cache Configuration Commands

Table 9-1 lists commands for configuring named data caches, for binding and unbinding objects to caches, and for reporting on cache bindings. It also lists procedures that you might use to check the size of your database objects, and commands that control cache usage at the object, command, or session level.

Table 9-1: Procedures and commands for using named caches

Command	Function
sp_cacheconfig	Creates or drops named caches, and changes the size or cache type.
sp_poolconfig	Creates and drops I/O pools, and changes their size.
sp_bindcache	Binds databases or database objects to a cache.
sp_unbindcache	Unbinds specific objects or databases from a cache.
sp_unbindcache_all	Unbinds all objects bound to a specified cache.

Table 9-1: Procedures and commands for using named caches (continued)

Command	Function
sp_helpcache	Reports summary information about data caches and lists the databases and database objects that are bound to caches.
sp_cachestrategy	Reports on cache strategies set for a table or index, and disables or re-enables prefetching or MRU strategy.
sp_logiosize	Changes the default I/O size for the log.
sp_spaceused	Provides information about the size of tables and indexes or the amount of space used in a database.
sp_estspace	Estimates the size of tables and indexes, given the number of rows the table will contain.
sp_help	Reports which cache a table is bound to.
sp_helpindex	Reports which cache an index is bound to.
sp_helpdb	Reports which cache a database is bound to.
set showplan on	Reports on I/O size and cache utilization strategies for a query.
set statistics io on	Reports number of reads performed for a query.
set prefetch [on |off]	Enables or disables prefetching for an individual session.
select... (prefetch...lru | mru)	Forces the server to use the specified I/O size or MRU replacement strategy.

In addition to the commands to configure named data caches interactively, you can also use the configuration file. See "Configuring Data Caches with the Configuration File" on page 9-27.

Information on Data Caches

The system procedure **sp_cacheconfig** creates and configures named data caches. To see the size of the default cache, type:

```
sp_cacheconfig "default data cache"
```

```
Cache Name          Status    Type     Config Value Run Value
------------------ --------- -------- ------------ ------------
default data cache  Active    Default       0.00 Mb     59.36 Mb
                                       ------------ ------------
                                Total       0.00 Mb     59.36 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
   Config Size: 0.00 Mb,   Run Size: 59.36 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ----------- -----------
    2 Kb    512 Kb      0.00 Mb    59.36 Mb
```

Summary information for each cache is printed in a block at the top of the report, ending with a total size for all configured caches. Then, for each cache, there is a block of information reporting the configuration for the memory pools in the cache.

The meanings of the columns in the block of output describing caches are:

- “Cache Name” gives the name of the cache.
- “Status” indicates whether the cache is active or not. Possible values are:
  - “Pend / Act” – the cache was just created and will be active after a restart.
  - “Active” – the cache is currently active.
  - “Pend / Del” – the cache is active, but will be deleted at the next restart of the server. The cache size was reset to 0 interactively.
- “Type” indicates whether the cache can store data and log pages (“Mixed”) or log pages only (“Log Only”). Only the default cache has the type “Default.” You cannot change the type of the default data cache or change the type of any other cache to “Default.”
- “Config Value” displays the size of the cache after the next restart of SQL Server. In this case, the default data cache has not been explicitly configured, so its size is 0.
- “Run Value” displays the size that SQL Server is currently using. For the default data cache, this size will always be all data cache space that is not explicitly configured to another cache.

The second block of output begins with two lines of information that describe the cache, repeating the information from the first block. It then provides information for each pool in the cache:

- “IO Size” shows the size of the buffers in the pool. When you first configure a cache, all the space is assigned to the 2K pool. Valid sizes are 2K, 4K, 8K, and 16K.
- “Wash Size” indicates the wash size for the pool. See “Changing the Wash Area for a Memory Pool” on page 9-18.
- “Config Size” and “Run Size” display the configured size and the size currently in use. These differ for the 2K pool because you cannot explicitly configure its size. These may differ for other pools if you have tried to move space between them, and some of the space could not be freed.

A summary line prints the total size of the cache or caches displayed.

Configuring Data Caches

After all other memory needs on SQL Server have been satisfied, all remaining space is available for the data cache. The first step in planning cache configuration and implementing caches is to set the **total memory** configuration parameter. After you set the configuration parameter and restart SQL Server, you can see exactly how much space is available for data caches on your server. For an overview of SQL Server memory usage, see Chapter 8, “Configuring Memory.”

You can configure data caches in two ways:

- Interactively, using **sp_cacheconfig** and **sp_poolconfig**
- By editing your configuration file

The following sections describe the process of cache configuration using **sp_cacheconfig** and **sp_poolconfig**. See “Configuring Data Caches with the Configuration File” on page 9-27 for information about cache configuration using the configuration file.

Each time you execute **sp_cacheconfig** or **sp_poolconfig**, SQL Server writes the new cache or pool information into the configuration file, and copies the old version of the file to a backup file. A message giving the backup file name is sent to the errorlog.

The syntax to create a new cache is:

```
sp_cacheconfig cache_name, "size[P|K|M|G]"
```

Size units can be specified with “P” for pages, “K” for kilobytes, “M” for megabytes, or “G” for gigabytes. The default is “K.” Maximum data cache size is limited only by the amount of memory available on your system.

This command configures a 10MB cache named *pubs_cache*:

```
sp_cacheconfig pubs_cache, "10M"
```

This command makes changes in the system tables and writes the new values to the configuration file, but does not activate the cache. You must restart SQL Server for the changes to take effect.

Using **sp_cacheconfig** to see the configuration before a restart shows different "Config" and "Run" values:

```
sp_cacheconfig "pubs_cache"
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_cache          Pend/Act  Mixed        10.00 Mb      0.00 Mb
                                       ------------ ------------
                                Total      10.00 Mb      0.00 Mb
```

The status "Pend / Act" for *pubs_cache* shows that the configuration of this cache is pending, waiting upon a restart. "Config Value" displays 10MB, while "Run Value" displays the value 0. Run values and configuration values are also different when you delete caches and when you change their size.

The section of output that provides detail about pools is not printed for caches that are not active.

After a restart of SQL Server, you see:

```
sp_cacheconfig
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
default data cache  Active    Default       0.00 Mb     49.28 Mb
pubs_cache          Pend/Act  Mixed        10.00 Mb     10.00 Mb
                                       ------------ ------------
                                Total      10.00 Mb     59.28 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
   Config Size: 0.00 Mb,   Run Size: 49.28 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb    49.28 Mb
================================================================
Cache: pubs_cache,   Status: Active,   Type: Mixed
   Config Size: 10.00 Mb,   Run Size: 10.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb    10.00 Mb
```

The *pubs_cache* is now active, and all the space is assigned to the 2K pool. The size of the default cache has been reduced by 10MB. The remainder of the difference in the size of the default cache and the total amount of cache available is due to changing overhead values. See “How Overhead Affects Total Cache Space” on page 9-16 for examples.

You can create as many caches as you want before the restarting SQL Server. You must restart SQL Server before you can configure pools or bind objects to newly created caches.

Explicitly Configuring the Default Cache

If you want to “lock in” some portion of the cache space for the default data cache, you can execute **sp_cacheconfig** with **default data cache** and a size value. This command ensures that no other cache configuration commands reduce the size of the default cache to less than 25MB:

```
sp_cacheconfig "default data cache", "25M"
```

After a restart of the server, the “Config Value” column shows these values:

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
default data cache  Active    Default      25.00 Mb     49.28 Mb
pubs_cache          Pend/Act  Mixed        10.00 Mb      0.00 Mb
                                       ------------ ------------
                                Total      35.00 Mb     49.28 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
   Config Size: 0.00 Mb,   Run Size: 49.28 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb     49.28 Mb
```

This command set a minimum size for the default data cache. You can change the minimum, but you cannot inadvertently allocate this space to other caches. With a minimum “Config Value” set, the “Run Value” still shows that the default data cache is allocated all of the memory that is not explicitly allocated to other caches.

Changing a Cache’s Type

If you want to reserve a cache for use only by the transaction log, you change the cache’s type to “logonly”. The following example creates the cache *pubs_log* with the type “logonly”:

```
sp_cacheconfig pubs_log, "7M", "logonly"
```

The following shows its state before a restart:

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_log            Pend/Act  Log Only      7.00 Mb      0.00 Mb
                                       ------------ ------------
                                 Total      7.00 Mb      0.00 Mb
```

You can change the type of an existing “mixed” cache, as long as there are no non-log objects bound to it:

```
sp_cacheconfig pubtune_cache, logonly
```

➤ ***Note***

In high transaction environments, SQL Server has performed best with a 4K pool configured for the transaction log. For information on configuring caches for improved log performance, see “Matching Log I/O Size for Log Caches” on page 9-12.

Dividing a Data Cache into Memory Pools

After you create a data cache, you can divide it into memory pools, each with a different I/O size. In any cache, you can have only one pool of each I/O size.

When SQL Server performs large I/Os, multiple pages are read into the cache at once. These pages are always treated as a unit: they age in the cache and are written to disk as a unit.

By default, when you create a named data cache, all its space is assigned to the default 2K memory pool. Creating additional pools reassigns some of that space to other pools, reducing the size of the 2K pool. For example, if you create a data cache with 50MB of space, all the space is assigned to the 2K pool. If you configure a 4K pool with 30MB of space in this cache, the 2K pool is reduced to 20MB.

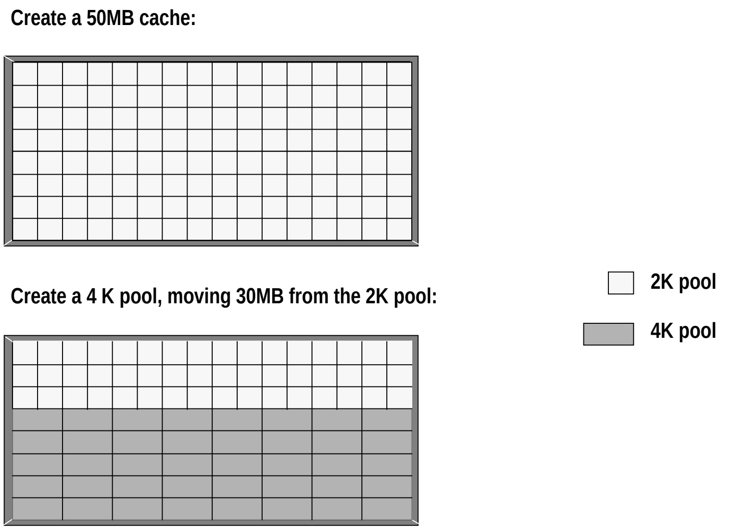


Figure 9-2: Configuring a cache and a 4K memory pool

Once you create pools, you can move space between them. In a cache with a 20MB 2K pool, and a 30MB 4K pool, you can configure a 16 K pool, taking 10MB of space from the 4K pool.

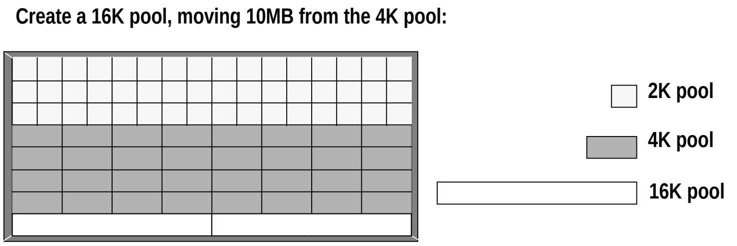


Figure 9-3: Moving space from an existing pool to a new pool

The commands that move space between pools within a cache do not require a restart of SQL Server to take effect, so you can reconfigure pools to meet changing application loads with little impact on server activity.

In addition to creating pools in the caches you configure, you can also add memory pools for I/Os up to 16K to the default data cache.

The syntax for configuring memory pools is:

```
sp_poolconfig cache_name, "memsize[P|K|M|G]",
   "config_poolK" [, "affected_poolK"]
```

Pool configuration always configures the *config_pool* to the size specified in the command. It always affects a second pool (the *affected_pool*) by moving space to or from that pool. If you do not specify the *affected_pool*, the space is taken from or allocated to the 2K pool. The minimum size for a pool is 512K.

Pool configuration is dynamic. You do not need to restart SQL Server for it to take effect.

This example creates a 7MB pool of 16K pages in the *pubs_cache* data cache:

```
sp_poolconfig pubs_cache, "7M", "16K"
```

This command reduces the size of the 2K memory pool. To see the current configuration, run **sp_cacheconfig**, giving only the cache name:

```
sp_cacheconfig pubs_cache
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_cache          Active    Mixed       10.00 Mb     10.00 Mb
                                       ------------ ------------
                              Total       10.00 Mb     10.00 Mb
=================================================================
Cache: pubs_cache,   Status: Active,   Type: Mixed
   Config Size: 10.00 Mb,   Run Size: 10.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb      3.00 Mb
   16 Kb   1424 Kb      7.00 Mb      7.00 Mb
```

You can also create memory pools in the default data cache. Starting with this configuration:

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
default data cache  Active    Default      25.00 Mb     42.21 Mb
                                       ------------ -----------
                                 Total      25.00 Mb     42.21 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
   Config Size: 25.00 Mb,   Run Size: 42.21 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb     42.21 Mb
```

This command creates a 16K pool:

```
sp_poolconfig "default data cache", "8M", "16K"
```

It results in this configuration, reducing the “Run Size” of the 2K pool:

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
default data cache  Active    Default      25.00 Mb     42.21 Mb
                                       ------------ -----------
                                 Total      25.00 Mb     42.21 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
   Config Size: 25.00 Mb,   Run Size: 42.21 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb     34.21 Mb
   16 Kb   4096 Kb      8.00 Mb      8.00 Mb
```

You do not need to configure the size of the 2K memory pool in cache. Its “Run Size” represents all of the memory not explicitly configured to other pools in the cache.

Matching Log I/O Size for Log Caches

If you create a cache for the transaction log of a database, configure most of the space in that cache to match the log I/O size. The default value is 4K, but SQL Server uses 2K I/O for the log if there is not a 4K pool available. The log I/O size can be changed with the system procedure **sp_logiosize**. Log I/O size for each database is reported in the error log when SQL Server starts, or you can check the size for a database by using the database and issuing **sp_logiosize** with no parameters.

This example configures the *pubs_log* cache for 4K I/O:

```
sp_poolconfig pubs_log, "3M", "4K"
```

You can also create a 4K memory pool in the default data cache for use by transaction logs of any databases that are not bound to another cache:

```
sp_poolconfig "default data cache", "2.5M", "4K"
```

Binding Objects to Caches

The system procedure **sp_bindcache** assigns a database, table, index or text/image object to a cache. In order to bind an entity to a cache:

- The named cache must exist, and its status must be "Active."
- The database or database object must exist.
- To bind tables, indexes, or objects, you must be using the database where they are stored.
- To bind system tables, including the transaction log table *syslogs*, the database must be in single-user mode.
- To bind a database, you must be using *master*, and the database must be in single user mode.
- The type of the cache must be "Mixed" to bind a database, user table, index, text or image object to a cache. Only the *syslogs* table can be bound to a cache of "Log Only" type.
- You must own the object or be the Database Owner or the System Administrator.

You must restart SQL Server after creating caches in order to bind objects to them. Bindings take effect immediately and do not require a restart.

The syntax for binding objects to caches is:

```
sp_bindcache cache_name, dbname [,[owner.]tablename
   [, indexname |  "text only" ] ]
```

The owner name is optional if the table is owned by "dbo".

This command binds the *titles* table to the *pubs_cache*:

```
sp_bindcache  pubs_cache, pubs2, titles
```

To bind an index on *titles*, add the index name as the third parameter:

```
sp_bindcache pubs_cache, pubs2,  titles, titleind
```

The owner name is not needed in the examples above because the *titles* table is owned by "dbo". To specify a table owned by any other user, add the owner name.You must enclose the parameter in quotation marks, since the period in the parameter is a special character:

```
sp_bindcache pubs_cache, pubs2, "fred.sales_east"
```

This command binds the transaction log, *syslogs*, to the *pubs_log* cache:

```
sp_bindcache pubs_log, pubs2, syslogs
```

The database must be in single-user mode in order to bind any system tables, including the transaction log, *syslogs*, to a cache. Use the **sp_dboption** system procedure from *master*, and a **use *database*** command, and run **checkpoint**:

```
sp_dboption pubs2, single, true
use pubs2
checkpoint
```

text and *image* columns for a table are stored in a separate data structure in the database. To bind this object to a cache, add the "text" parameter:

```
sp_bindcache pubs_cache, pubs2, au_pix,
"text only"
```

This command, executed from *master*, binds the *tempdb* database to a cache:

```
sp_bindcache tempdb_cache, tempdb
```

You can rebind objects without dropping existing bindings.

Cache Binding Restrictions

You cannot bind or unbind a database object:

- When dirty reads are active on the object
- When a cursor is open on the object

In addition, SQL Server needs to lock the object while the binding or unbinding takes place, so the command may have slow response time. See "Locking to Perform Bindings" on page 9-27 for more information.

Getting Information About Cache Bindings

The **sp_helpcache** system procedure provides information about a cache and the entities bound to it when you provide the cache name:

```
     sp_helpcache pubs_cache
Cache Name         Config Size     Run Size       Overhead
----------------- -------------   ----------     ----------
pubs_cache          10.00 Mb      10.00 Mb        0.53 Mb

-------------- Cache Binding Information: --------------

Cache Name   Entity Name       Type      Index Name   Status
----------   -----------       ----       ---------   ------
pubs_cache   pubs2.dbo.titles  index     titleind     V
pubs_cache   pubs2.dbo.au_pix  index     tau_pix      V
pubs_cache   pubs2.dbo.titles  table                  V
```

If you use **sp_helpcache** without a cache name, it prints information about all the configured caches on SQL Server and all the objects that are bound to them.

sp_helpcache performs string matching on the cache name, using "%*cachename*%". So, for example, "pubs" matches both "pubs_cache" and "pubs_log".

The "Status" column reports whether a cache binding is valid (”V”) or invalid ("I"). If an database or object is bound to a cache, and the cache is deleted, binding information is retained in the system tables, but the cache binding is marked as invalid. All objects with invalid bindings use the default data cache. If you subsequently create another cache with the same name, the binding becomes valid when the cache is activated by a re-start of SQL Server.

Checking Cache Overhead

sp_helpcache can report the amount of overhead required to manage a named data cache of a given size. When you create a named data cache, all the space you request with **sp_cacheconfig** is made available for cache space. The memory needed for cache management is taken from the default data cache.

To see the overhead required for a cache, give the proposed size. You can use the units "P" for pages, "K" for kilobytes, "M" for megabytes, or "G" for gigabytes. The following examples check the overhead for 20,000 pages:

```
sp_helpcache "20000P"
```

```
2.08Mb of overhead memory will be needed to manage
a cache of size 20000P
```

Note that you are not "wasting" any cache space by configuring user caches. About 5 percent of memory is required for the structures that store and track pages in memory, whether you use a single large data cache or several smaller caches.

How Overhead Affects Total Cache Space

On page 9-7, the example shows a default data cache with 59.36MB of cache space available before any user-defined caches are created. When the 10MB *pubs_cache* is created and SQL Server is restarted, the results of **sp_cacheconfig** show a total cache size of 59.28MB.

The process of configuring a data cache can appear to increase or decrease the total available cache. The explanation for this lies in the amount of overhead required to manage a cache of a particular size, and the fact that the overhead is not included in the values displayed by **sp_cacheconfig**.

Using **sp_helpcache** to check the overhead of the original 59.36MB default cache and the new 10MB cache shows that the change in space is due to changes in the size of overhead. The following command shows the overhead for the default data cache before any changes were made:

```
sp_helpcache "59.36M"
```

```
3.03Mb of overhead memory will be needed to manage
a cache of size 55.96M
```

This command shows the overhead for *pubs_cache*:

```
sp_helpcache "10M"
```

```
0.53Mb of overhead memory will be needed to manage
a cache of size 10M
```

The following calculations add the overhead required to manage the original cache space and then subtract the space the overhead for *pubs_cache*.

Original total cache size (overhead not included)	59.36
Overhead for 59.36MB default cache	+3.03
Total cache space, including overhead	62.39
Subtract 10MB *pubs_cache* and .53MB overhead	- 10.53
Remaining space	51.86
Overhead for 51.86MB cache	- 2.86
Usable size of the default cache	49.30

Cache sizes are rounded to two places when printed by **sp_cacheconfig**, and overhead is rounded to two places by **sp_helpcache**, so you will see a small amount of rounding error in the output.

Dropping Cache Bindings

Two commands drop cache bindings:

- **sp_unbindcache** unbinds a single entity from a cache
- **sp_unbindcache_all** unbinds all objects bound to a cache

The syntax for **sp_unbindcache** is:

```
sp_unbindcache dbname [,[owner.]tablename
   [, indexname | "text only"] ]
```

This command unbinds the *pubs2* database:

```
sp_unbindcache pubs2
```

This command unbinds the *titles* table:

```
sp_unbindcache pubs2, titles
```

This commands unbinds the *titleidind* index:

```
sp_unbindcache pubs2, titles, titleidind
```

In order to unbind all the objects bound to a cache, use **sp_unbindcache_all**, giving the cache’s name:

```
sp_unbindcache_all pubs_cache
```

You cannot use **sp_unbindcache_all** if more than 8 databases and/or objects in 8 databases are bound to the cache. You must use

sp_unbindcache on individual databases or objects to reduce the number of databases involved to 8 or less.

When you drop a cache binding for an object, all the pages currently in the memory are cleared from the cache.

Changing the Wash Area for a Memory Pool

SQL Server attempts to ensure that queries that need clean pages in a data cache find them at the LRU (least recently used) end of each memory pool. A portion of the pool is configured as the **wash area**. Once dirty pages (pages that have been changed in cache) pass the wash marker and enter the wash area, SQL Server starts an asynchronous I/O on the page. When the write completes, the page is marked clean and remains available in the cache until it reaches the LRU.

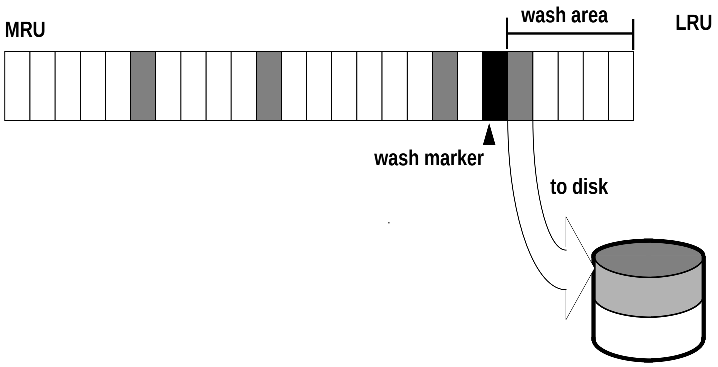


Figure 9-4: Wash area of a buffer pool

By default, the size of the wash area for a memory pool is configured to be the smaller of:

- 512K
- 20 percent of the buffers in the pool, down to a minimum of 10 buffers

The maximum size of the wash area is 80 percent of the pool size.

A buffer is a block of pages matching the I/O size for the pool. Each buffer is treated as a unit: all pages in the buffer are read into cache, written to disk, and aged in the cache as a unit. For a 2K pool, 256 buffers equals 512K; for a 16K pool, 256 buffers equals 4096K.

For example, if you configure a 16K pool with 1MB of space, the pool has 64 buffers, 20 percent is 12.8. This is rounded down, so 12 buffers, or 192K is allocated to the wash area.

If your cache is very large, and if you have a high rate of updates on your system, you may need to change the wash size to 1 or 2 percent of the pool size. The next section explains performance impacts of wash size configuration.

When the Wash Area Is Too Small

If the wash area is too small for the usage in a buffer pool, operations needing a clean buffer may have to wait for I/O to complete on the dirty buffer at the LRU end of the pool. This is called a "dirty buffer grab," and it can seriously slow performance. Figure 9-5 shows a dirty buffer grab.

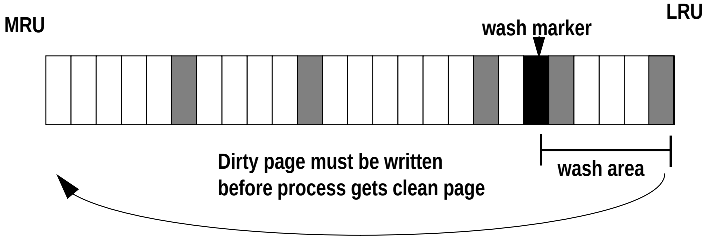


Figure 9-5: Small wash area results in a dirty buffer grab

You can use the system procedure **sp_sysmon** to determine if dirty buffer grabs are taking place in your memory pools. Run **sp_sysmon** while the cache is experiencing a heavy period of I/O and heavy update activity, since it is the combination of many dirty pages and high cache usage that usually causes dirty buffer grabs.

If the "Buffers Grabbed Dirty" output in the cache summary section shows a non-zero value in the "Count" column, check the "Grabbed Dirty" row for each pool to determine where the problem lies. Increase the size of the wash area for the affected pool. This command sets the wash area of the 2K memory pool to 720K:

```
sp_poolconfig pubs_cache, "2K", "wash=720K"
```

If the pool is very small, you may also want to increase the pool size, especially if **sp_sysmon** output shows that the pool is experiencing high turnover rates.

For more information, see “Grabbed Dirty” on page 19-57 of the *Performance and Tuning Guide.*

When the Wash Area Is Too Large

If the wash area is too large, the buffers move too quickly past the “wash marker” in cache, and an asynchronous write is started on the buffer, as shown in Figure 9-6. The buffer is marked “clean” and remains in the wash area of the MRU/LRU chain until it reaches the LRU. If another query changes a page in the buffer, SQL Server must perform additional I/O to write it to disk again.

If **sp_sysmon** output shows a high percentage of buffers “Found in Wash,” and there are no problems with dirty buffer grabs, you may want to try reducing the size of the wash area. See “Found in Wash” on page 19-55 of the *Performance and Tuning Guide* for more information.

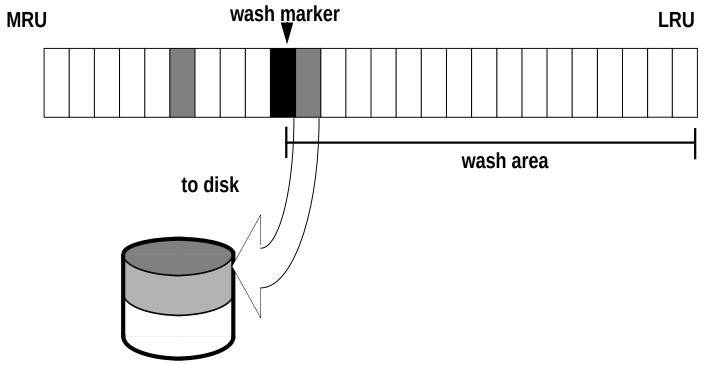


Figure 9-6: Effects of making the wash area too large

Resizing Named Data Caches

To change the size of an existing cache, issue **sp_cacheconfig**, specifying a new total size for the cache. When you increase the size of a cache by specifying a larger size with **sp_cacheconfig**, all the additional space is added to the 2K pool. When you decrease the size of a cache, all the space is taken from the 2K pool. You cannot decrease the size of the 2K pool to less than 512K.

Increasing the Size of a Cache

sp_cacheconfig reports that *pubs_cache* is currently configured with 10MB of space:

```
    sp_cacheconfig pubs_cache
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_cache          Active    Mixed        10.00 Mb     10.00 Mb
                                       ------------ ------------
                                Total      10.00 Mb     10.00 Mb
================================================================
Cache: pubs_cache,   Status: Active,   Type: Mixed
   Config Size: 10.00 Mb,   Run Size: 10.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    720 Kb      0.00 Mb      3.00 Mb
    4 Kb   1024 Kb      4.00 Mb      4.00 Mb
   16 Kb   1424 Kb      3.00 Mb      3.00 Mb
```

If you want to increase the size of the cache and its 2K pool, specify the new total size of the cache:

```
    sp_cacheconfig pubs_cache, "20M"
```

The following reports the configuration before a restart:

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_cache          Active    Mixed        20.00 Mb     20.00 Mb
                                       ------------ ------------
                                Total      20.00 Mb     20.00 Mb
================================================================
Cache: pubs_cache,   Status: Active,   Type: Mixed
   Config Size: 20.00 Mb,   Run Size: 20.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb     13.00 Mb
    4 Kb   1024 Kb      4.00 Mb      4.00 Mb
   16 Kb   1424 Kb      3.00 Mb      3.00 Mb
```

The additional 10MB has been configured and becomes available in the 2K pool at the next restart.

Decreasing the Size of a Cache

You can also reduce the size of a cache. For example, the following is a report on the *pubs_log* cache:

```
sp_cacheconfig pubs_log
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_log            Active    Log Only      7.00 Mb      7.00 Mb
                                       ------------ ------------
                                Total       7.00 Mb      7.00 Mb
================================================================
Cache: pubs_log,   Status: Active,   Type: Log Only
   Config Size: 7.00 Mb,   Run Size: 7.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb      4.00 Mb
    4 Kb   1024 Kb      3.00 Mb      3.00 Mb
```

The following command reduces the size of the *pubs_log* cache, reducing the size of the 2K pool:

```
sp_cacheconfig pubs_log, "6M"
```

After a re-start, **sp_cacheconfig** shows:

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_log            Active    Log Only      6.00 Mb      6.00 Mb
                                       ------------ ------------
                                Total       6.00 Mb      7.00 Mb
================================================================
Cache: pubs_log,   Status: Active,   Type: Log Only
   Config Size: 6.00 Mb,   Run Size: 7.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb      3.00 Mb
    4 Kb   1024 Kb      3.00 Mb      3.00 Mb
```

When you reduce the size of a data cache, all the space to be removed must be available in the 2K pool. You may need to move space to the 2K pool from other pools before you can reduce the size of the data cache. In the last example, if you wanted to reduce the size of the cache to 3MB, you would need to use **sp_poolconfig** to move some memory into the 2K pool from the 4K pool. See “Changing the Size of Memory Pools” on page 9-23 for more information.

Dropping Data Caches

To completely remove a data cache, reset its size to 0:

```
sp_cacheconfig pubs_log, "0"
```

This changes the cache status to "Pend/Del". You must restart SQL Server for the change to take effect. Until you do, the cache remains active, and all objects bound to the cache still use it for I/O.

If you delete a data cache, and there are objects bound to the cache, the cache bindings are marked invalid at the next restart of SQL Server. All objects with invalid cache bindings use the default data cache. Warning messages are printed in the error log when the bindings are marked invalid. If you re-create the cache and restart SQL Server, the bindings are marked valid again.

You cannot drop the default data cache.

Changing the Size of Memory Pools

To change the size of a memory pool, use **sp_poolconfig** to specify the cache, the new size for the pool, the I/O size of the pool you want to change, and the I/O size of the pool from which the buffers should be taken. If you do not specify the final parameter, all the space is taken from or assigned to the 2K pool.

Moving Space from the 2K Memory Pool

This command checks the current configuration of the *pubs_log* cache:

```
sp_cacheconfig pubs_log
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_log            Active    Log Only      6.00 Mb      6.00 Mb
                                       ------------ ------------
                                Total       6.00 Mb      6.00 Mb
================================================================
Cache: pubs_log,   Status: Active,   Type: Log Only
   Config Size: 6.00 Mb,   Run Size: 7.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb      3.00 Mb
    4 Kb   1024 Kb      3.00 Mb      3.00 Mb
```

This command increases the size of the 4K pool to 5MB, moving the required space from the 2K pool:

```
sp_poolconfig pubs_log, "5M", "4K"
sp_cacheconfig pubs_log
```

```
Cache Name          Status    Type     Config Value Run Value
------------------- --------- -------- ------------ ------------
pubs_log            Active    Log Only      6.00 Mb      6.00 Mb
                                       ------------ ------------
                                Total       6.00 Mb      6.00 Mb
================================================================
Cache: pubs_log,   Status: Active,   Type: Log Only
   Config Size: 6.00 Mb,   Run Size: 7.00 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb    512 Kb      0.00 Mb      1.00 Mb
    4 Kb   1024 Kb      5.00 Mb      5.00 Mb
```

Moving Space from Other Memory Pools

To transfer space from a pool other than the 2K pool, you specify a “to” I/O size and a “from” I/O size. This output shows the current configuration of the default data cache:

```
Cache Name           Status    Type     Config Value Run Value
-------------------- --------- -------- ------------ ------------
default data cache   Active    Default      25.00 Mb     43.21 Mb
                                        ------------ ------------
                                 Total      25.00 Mb     43.21 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
     Config Size: 25.00 Mb,   Run Size: 43.21 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb   8336 Kb      0.00 Mb     32.71 Mb
    4 Kb   1024 Kb      2.50 Mb      2.50 Mb
   16 Kb   4096 Kb      8.00 Mb      8.00 Mb
```

The following command increases the size of the 4K pool from 2.5MB to 4MB, taking the space from the 16K pool:

```
sp_poolconfig "default data cache","4M", "4K","16K"
```

This command results in the following configuration:

```
Cache Name           Status    Type     Config Value Run Value
-------------------- --------- -------- ------------ ------------
default data cache   Active    Default      25.00 Mb     43.21 Mb
                                        ------------ ------------
                                  Total     25.00 Mb     43.21 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
     Config Size: 25.00 Mb,   Run Size: 43.21 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb   8336 Kb      0.00 Mb     32.71 Mb
    4 Kb   1024 Kb      4.00 Mb      4.00 Mb
   16 Kb   4096 Kb      6.50 Mb      6.50 Mb
```

➤ ***Note***

You cannot reduce the size of the 2K pool to less than 512K.

When you issue a command to move buffers between pools in a cache, SQL Server can only move “free” buffers. It cannot move buffers that are in use or buffers that contain changes that have not been written to disk.

When SQL Server cannot move as many buffers as you request, it displays an informational message, giving the requested size and the resulting size of the memory pool.

Dropping a Memory Pool

To completely remove a pool, reset its size to 0. This command removed the 16K pool, placing all of the space in the 2K pool:

```
sp_poolconfig "default data cache", "0",  "16K"
sp_cacheconfig "default data cache"
```

```
Cache Name           Status    Type     Config Value Run Value
-------------------- --------- -------- ------------ ------------
default data cache   Active    Default      25.00 Mb     43.21 Mb
                                        ------------ ------------
                                 Total      25.00 Mb     43.21 Mb
================================================================
Cache: default data cache,   Status: Active,   Type: Default
     Config Size: 25.00 Mb,   Run Size: 43.21 Mb

IO Size  Wash Size Config Size  Run Size
-------- --------- ------------ ------------
    2 Kb   8336 Kb      0.00 Mb     39.21 Mb
    4 Kb   1024 Kb      4.00 Mb      4.00 Mb
```

If you do not specify the *affected_pool*, all the space is placed in the 2K pool. You cannot delete the 2K pool in any cache.

If the pool you are trying to delete contains pages that are in use, or pages that have been dirtied but not written to disk, SQL Server moves as many pages as possible to the specified pool and prints an informational message telling you the size of the remaining pool. The pages that cannot be moved are linked to transactions in progress. When these transactions complete and the pages are freed, you can reissue the command.

Cache Binding Effects on Memory and Query Plans

Binding and unbinding objects may have an impact on performance. When you bind or unbind a table or index:

- The object’s pages are flushed from the cache
- The object must be locked to perform the binding
- All query plans for procedures and triggers must be recompiled

Flushing Pages from Cache

When you bind an object or database to a cache, all the object’s pages that are already in memory are removed from the source cache. The next time the pages are needed by a query, they are read into the new cache. Similarly, when you unbind objects, the pages in cache are removed from the user-configured cache and read into the default cache the next time they are needed by a query.

Locking to Perform Bindings

In order to bind or unbind user tables, indexes, or text or image objects, the cache binding commands need to acquire an exclusive table lock on the object. If a user holds locks on a table, and you issue an **sp_bindcache**, **sp_unbindcache**, or **sp_unbindcache_all** on the object, the system procedure sleeps until it can acquire the locks it needs.

For databases, system tables, and indexes on system tables, the database must be in single-user mode, so there cannot be another user who holds a lock on the object.

Cache Binding Effects on Stored Procedures and Triggers

Cache bindings and I/O sizes are part of the query plan for stored procedures and triggers. When you change the cache binding for an object, all the stored procedures that reference the object are recompiled the next time they are executed. When you change the cache binding for a database, all stored procedures that reference any objects in the database that are not explicitly bound to a cache are recompiled the next time they are run.

Configuring Data Caches with the Configuration File

You can add or drop named data caches and reconfigure existing caches and their memory pools by editing the configuration file that is used when you start SQL Server.

➤ ***Note***

You cannot reconfigure caches and pools on a server while it is running by reading in a configuration file with **sp_configure**. Any attempt to read a configuration file that contains cache and pool configurations different than those already configured on the server causes the read to fail with an advisory message.

Cache and Pool Entries in the Configuration File

Each configured data cache on the server has a block of information in the configuration file:

```
[Named Cache:cache_name]
     cache size = {size | DEFAULT}
     cache status = {mixed cache | log only | default data cache}
```

The *size* specification can include “P” for pages, “K” for kilobytes, “M” for megabytes, or “G” for gigabtyes. If the units are not indicated, the default is “K”.

This example shows configuration file entry for the default data cache:

```
[Named Cache:default data cache]
        cache size = DEFAULT
        cache status = default data cache
```

The default data cache entry is the only cache entry that is absolutely required in order for SQL Server to start. It must have the cache size and cache status, and the status must be “default data cache.”

If the cache has pools configured in addition to the 2K pool, the block in the example above is followed by a block of information for each pool:

```
[16K I/O Buffer Pool]
        pool size = size
        wash size = size
```

The following example shows output from **sp_cacheconfig**, followed by the configuration file entries that match this cache and pool configuration:

```
 Cache Name             Status    Type     Config Value Run Value
 ------------------- --------- -------- ------------ ------------
 default data cache  Active    Default       0.00 Mb     33.15 Mb
 pubs_cache          Active    Mixed        20.00 Mb     20.00 Mb
 pubs_log            Active    Mixed         6.00 Mb      6.00 Mb
                                        ------------ ------------
                                 Total      26.00 Mb     59.15 Mb
=================================================================
Cache: default data cache,   Status: Active,   Type: Default
      Config Size: 0.00 Mb,   Run Size: 33.15 Mb

 IO Size  Wash Size Config Size  Run Size
 -------- --------- ------------ ------------
     2 Kb    512 Kb      0.00 Mb     22.15 Mb
     4 Kb   1024 Kb      3.00 Mb      3.00 Mb
    16 Kb   4096 Kb      8.00 Mb      8.00 Mb
================================================================
Cache: pubs_cache,   Status: Active,   Type: Mixed
      Config Size: 20.00 Mb,   Run Size: 20.00 Mb
 IO Size  Wash Size Config Size  Run Size
 -------- --------- ------------ ------------
     2 Kb    512 Kb      0.00 Mb     16.00 Mb
     4 Kb    204 Kb      1.00 Mb      1.00 Mb
    16 Kb    608 Kb      3.00 Mb      3.00 Mb
=================================================================
Cache: pubs_log,   Status: Active,   Type: Mixed
      Config Size: 6.00 Mb,   Run Size: 6.00 Mb

 IO Size  Wash Size Config Size  Run Size
 -------- --------- ------------ ------------
     2 Kb    512 Kb      0.00 Mb      1.00 Mb
     4 Kb   1024 Kb      5.00 Mb      5.00 Mb
```

This is the matching configuration file information:

```
[Named Cache:default data cache]
        cache size = DEFAULT
        cache status = default data cache

[4K I/O Buffer Pool]
        pool size = 3.0000M
        wash size = 1024 K

[16K I/O Buffer Pool]
        pool size = 8.0000M
        wash size = 4096 K

[Named Cache:pubs_cache]
        cache size = 20M
        cache status = mixed cache

[4K I/O Buffer Pool]
        pool size = 1.0000M
        wash size = 204 K

[16K I/O Buffer Pool]
        pool size = 3.0000M
        wash size = 608 K

[Named Cache:pubs_log]
        cache size = 6M
        cache status = mixed cache

[4K I/O Buffer Pool]
        pool size = 5.0000M
        wash size = 1024 K
```

For more information about the configuration file, see Chapter 11, "Setting Configuration Parameters."

◆ ***WARNING!***

Be sure to check the total memory configuration parameter and allow enough memory for other SQL Server needs. If you attempt to assign too much memory to data caches in your configuration file, SQL Server will not start. If this occurs, edit the configuration file to reduce the amount of space in the data caches, or increase the total memory allocated to SQL Server.

Configuration File Errors

If you edit your configuration file by hand, check cache, pool, and wash sizes carefully. Certain configuration file errors can cause start-up failure:

- The total size of all of the caches cannot be greater than the amount of total memory, minus other SQL Server memory needs.
- The total size of pools in any cache cannot be greater than the size of the cache.
- The wash size cannot too small (less than 20% of the pool size, down to a minimum of 10 buffers) and cannot be larger than 80% of the buffers in the pool.
- The default data cache status must be "default data cache", and the size must be specified, either as a numeric value or as "DEFAULT".
- The status and size for any cache must be specified.
- The pool size and wash size for all pools larger than 2K must be specified.
- The status of all user-defined caches must be "mixed cache" or "log only".

In most cases, problems with missing entries are reported as "unknown format" errors on lines immediately after the entry where the size, status, or other information was omitted. Other errors provide the name of the cache where the error occurred, and the type of error. For example, you see this error if the wash size for a pool is specified incorrectly:

```
The wash size for the 4k buffer pool in cache
pubs_cache has been incorrectly configured. It
must be a minimum of 10 buffers and a maximum of
80 percent of the number of buffers in the pool.
```

Cache Configuration Guidelines

Additional information on cache configuration and performance and suggested strategies for testing cache utilization is contained in Chapter 15, “Memory Use and Performance” in the *Performance and Tuning Guide*. These are some general guidelines for System Administrators:

- Make sure that your default data cache is large enough for all cache activity on unbound tables and indexes. All objects that are not explicitly bound to a cache use the default cache. This includes any unbound system tables in user databases, the system tables in master, and any other objects that are not explicitly bound to a cache.
- During recovery, only the 2K memory pool of the default cache is active. Transactions logs are read into the 2K pool of the default cache. All transactions that must be rolled back or rolled forward must read data pages into the default data cache. If the default data cache is too small, it can slow recovery time.
- Do not “starve” the 2K pool in any cache. For many types of data access, there is no need for large I/O. For example, a simple query that uses an index to return a single row to the user might use four or five 2K I/Os, and gain nothing from 16K I/O.
- Certain commands can perform only 2K I/O: **disk init**, certain **dbcc** commands, and **drop table**. **dbcc checktable** can perform large I/O, and **dbcc checkdb** performs large I/O on tables, and 2K I/O on indexes.
- For caches used by transaction logs, configure an I/O pool that matches the default log I/O size. This size is set for a database using the system procedure **sp_logiosize**. The default value is 4K.
- Trying to micro-manage every index and object and its caching can waste cache space. If you have created caches or pools that are not optimally used by the tables or indexes bound to them, they are wasting space and creating additional I/O in other caches.

- If *tempdb* is heavily used by your applications, bind it to its own cache. Note that you can bind only the entire *tempdb* database, you cannot bind individual objects from *tempdb*.
- For very large caches with high update rates, be sure that your wash size is large enough.
- On multi-CPU systems, spread your busiest tables and their indexes across multiple caches to avoid spinlock contention.
- Consider reconfiguring caches or the memory pools within caches to match changing workloads. Reconfiguring caches requires a restart of the server, but memory pool reconfiguration does not.

  For example, if your system performs mostly OLTP (online transaction processing) during most of the month, and has heavy DSS (decision support system) activity for a few days, consider moving space from the 2K pool to the 16K pool for the high DSS activity, and resizing the pools for OLTP when the DSS workload ends.

10 Managing Multiprocessor Servers

This chapter provides guidelines for administering SQL Server on multiprocessor hardware. It includes the following sections:

Introduction

SQL Server implements the Sybase Virtual Server Architecture™, which enables 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 machine. 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

For information on application design for SMP systems, see Chapter 17, "Using CPU Resources Effectively,"in the *Performance and Tuning Guide*.

Definitions

Following 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 (**task affinity**), a certain engine handles network I/O for a certain task (**network I/O affinity**), or a certain engine runs only on a certain CPU (**engine affinity**).
- **Network affinity migration**: Describes the process of moving network I/O from one engine to another. SMP systems that support this migration allow SQL Server to distribute the network I/O load among all of its engines.

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 of which run the server program in parallel. See Figure 10-1.

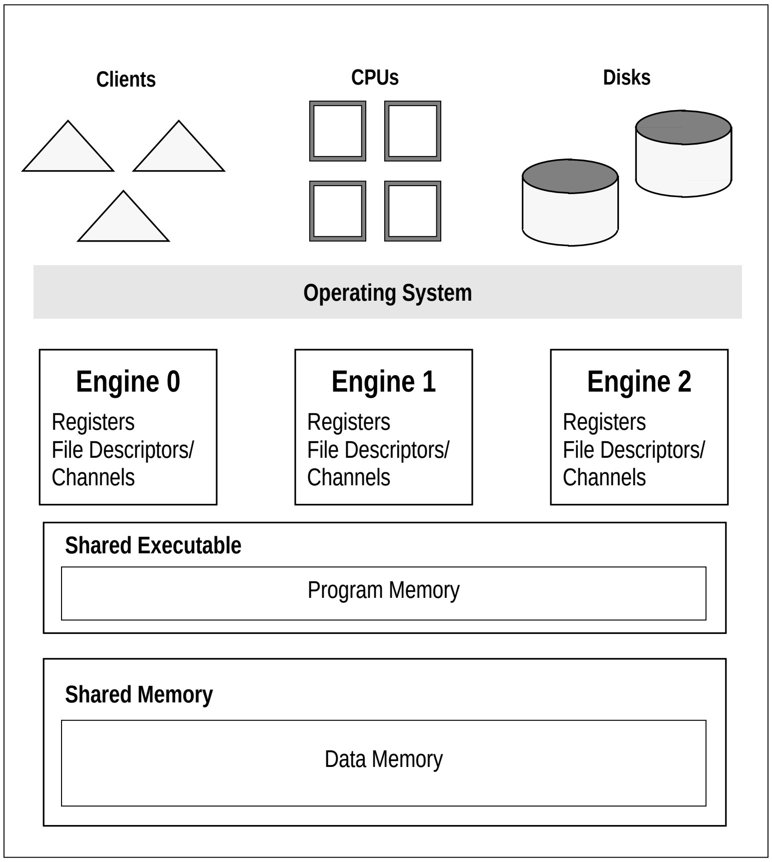


Figure 10-1: SMP environment architecture

If the SMP system supports **network affinity migration**, SQL Server migrates the client connection at login time to the engine that is currently servicing the smallest number of network I/O connections. Otherwise, only one of the engines, engine 0, handles the tasks involving network management. In other aspects, all engines are peers, and they 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 executes any runnable client task; there is no **task affinity**.

The operating system schedules the engine processes onto physical processors. Any available CPU is used for any engine; there is no **engine affinity**. 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 10-2: 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 (that is, 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.

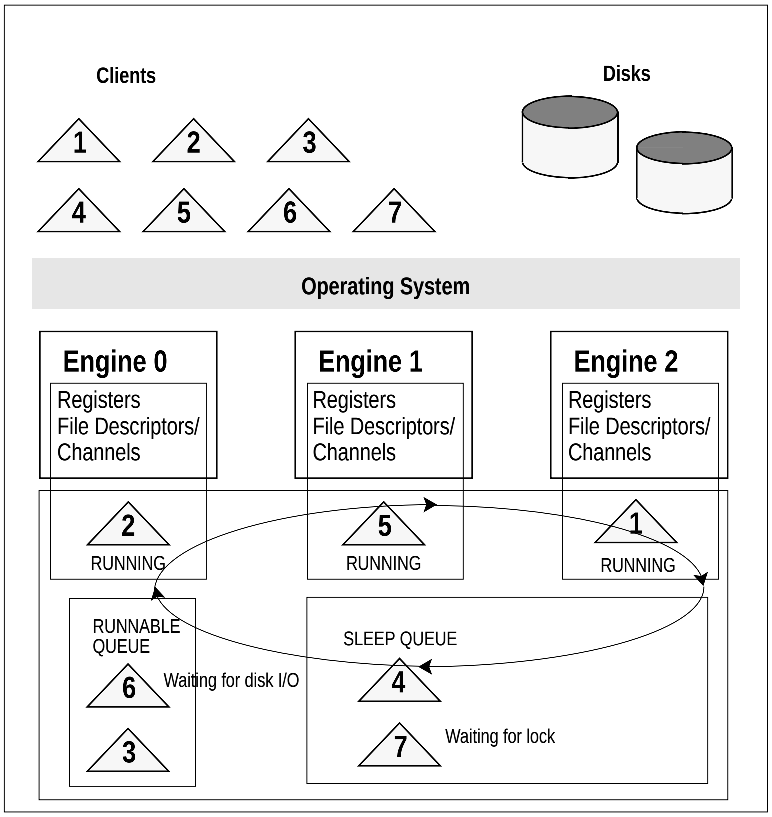


Figure 10-2: SQL Server task management in the SMP environment

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 more powerful 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.

However, increasing or decreasing engines is not a dynamic configuration, so you must restart the server to reset the number of engines.

The **max online engines** configuration parameter controls the number of engines that SQL Server uses. Reset this parameter 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. Stop and restart the server.

Repeat these steps whenever you need to change the number of engines. Engines other than engine 0 are brought online after recovery is complete.

See also the **min online engines** configuration parameter, which is described under “min online engines” on page 11-67.

Choosing the Right Number of Engines

It is important to choose the right number of engines for SQL Server. Here are some guidelines for choosing how many engines to use:

- **Never** have more engines than CPUs. Doing so may slow performance. If a CPU goes offline, use **sp_configure** to reduce the **max online engines** configuration parameter by one and restart SQL Server.
- Have only as many engines as **usable** CPUs. If there is a lot of processing by the client or other non-SQL 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 a few engines and add engines 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 SQL Server installation and configuration guide for the appropriate utility for your operating system.

Managing User Connections

If the SMP system supports network affinity migration, each engine handles the network I/O for its connections. During login, SQL Server migrates the client connection task from engine 0 to the engine currently servicing the smallest number of connections. The client's tasks run network I/O on that engine (**network affinity**) until the connection is terminated. To determine if your SMP system supports this migration, see the SQL Server installation and configuration guide.

By distributing the network I/O among its engines, SQL Server can handle more user connections. The per-process limit on the maximum number of open file descriptors no longer limits the number of connections. Adding more engines linearly increases the maximum number of file descriptors, as stored in the global variable *@@max_connections*.

As you increase the number of engines, SQL Server prints the increased *@@max_connections* value to standard output and the error log file after you restart the server. You can always query the value as follows:

```
select @@max_connections
```

This number represents the maximum number of file descriptors allowed by the operating system for your process, minus the following file descriptors used by SQL Server:

- One for each master network listener on engine 0 (one for every "master" line in the interfaces file entry for that SQL Server)
- One for each engine's standard output
- One for each engine's error log file
- Two for each engine's network affinity migration channel
- One per engine for configuration
- One per engine for the interfaces file
- One per engine for internal use

For example, if SQL Server is configured for one engine, and the value of *@@max_connections* equals 1019, adding a second engine increases the value of *@@max_connections* to 2039 (assuming only one master network listener).

You can configure **number of user connections** to take advantage of an increased *@@max_connections* limit. However, each time you decrease the number of engines using **max online engines**, you must also adjust the **number of user connections** value accordingly. Reconfiguring **max online engines** or **number of user connections** is not dynamic, so you must restart the server to change these configuration values. For information about configuring **number of user connections**, see Chapter 11, "Setting Configuration Parameters."

Managing Memory

The **total memory** configuration parameter may require special attention in SMP sites.

Not all platforms require a higher memory configuration parameter than before. If your platform does, the installed value of the **total memory** configuration parameter reflects this, so you may never need to adjust it. If error message 701:

```
There is insufficient memory to run this query
```

appears in the error log and at the client terminal, you may want to increase the amount of procedure cache available.

Configuring Server Behavior

11 Setting Configuration Parameters

SQL Server Configuration Parameters

The following table lists all the configuration parameters for SQL Server. Some of the parameter names have been changed in release 11.0; for this reason, the table is alphabetized by the old names. In addition, there are a number of new parameters, and these are listed at the end of the table.

Name In Previous Release	New Name
T1204 (trace flag)	print deadlock information, page 11-90
T1603 (trace flag)	allow sql server async i/o, page 11-26
T1610 (trace flag)	tcp no delay, page 11-56
T1611 (trace flag)	lock shared memory, page 11-63
additional network memory	additional network memory, page 11-62
allow updates	allow updates to system tables, page 11-68
audit queue size	audit queue size, page 11-69
calignment	memory alignment boundary, page 11-21
cclkrate	sql server clock tick length, page 11-96
cfgcprot	permission cache entries, page 11-104
cguardsz	stack guard size, page 11-105
cindextrips	number of index trips, page 11-22
cmaxnetworks	max number network listeners, page 11-53
cmaxscheds	i/o polling process count, page 11-79
cnalarm	number of alarms, page 11-84
cnblkio	disk i/o structures, page 11-27
cnlanginfo	number of languages in cache, page 11-32
cnmaxaio_engine	max async i/os per engine, page 11-57
cnmaxaio_server	max async i/os per server, page 11-58
cnmbox	number of mailboxes, page 11-86
cnmsg	number of messages, page 11-86
coamtrips	number of oam trips, page 11-23
cpreallocext	number of pre-allocated extents, page 11-88

Name In Previous Release	New Name
cpu flush	cpu accounting flush interval, page 11-70
cschedspins	runnable process search count, page 11-91
csortbufsize	number of sort buffers, page 11-89
csortpgcount	sort page count, page 11-95
ctimemax	cpu grace time, page 11-71
database size	default database size, page 11-72
default character set id	default character set id, page 11-31
default language	default language id, page 11-31
default network packet size	default network packet size, page 11-48
default sortorder id	default sortorder id, page 11-32
devices	number of devices, page 11-28
extent i/o buffers	number of extent i/o buffers, page 11-85
fillfactor	default fill factor percent, page 11-73
identity burning set factor	identity burning set factor, page 11-76
i/o flush	i/o accounting flush interval, page 11-78
language in cache	number of languages in cache, page 11-32
locks	number of locks, page 11-41
max online engines	max online engines, page 11-66
maximum network packet size	max network packet size, page 11-50
memory	total memory, page 11-64
min online engines	min online engines, page 11-67
mrstart	shared memory starting address, page 11-61
nested trigger	allow nested triggers, page 11-67
open databases	number of open databases, page 11-87
open objects	number of open objects, page 11-88
password expiration interval	“systemwide password expiration”
pre-read packets	remote server pre-read packets, page 11-55
procedure cache	procedure cache percent, page 11-24
recovery flags	print recovery information, page 11-17
recovery interval	recovery interval in minutes, page 11-18
remote access	allow remote access, page 11-47

Name In Previous Release	New Name
remote connections	number of remote connections, page 11-54
remote logins	number of remote logins, page 11-54
remote sites	number of remote sites, page 11-55
sql server code size	executable code size, page 11-47
stack size	stack size, page 11-108
tape retention	tape retention in days, page 11-20
time slice	time slice, page 11-97
upgrade version	upgrade version, page 11-100
user connections	number of user connections, page 11-101
New Parameters	
N/A	address lock spinlock ratio, page 11-33
N/A	configuration file, page 11-30
N/A	deadlock checking period, page 11-35
N/A	deadlock retries, page 11-36
N/A	event buffers per engine, page 11-74
N/A	freelock transfer block size, page 11-38
N/A	housekeeper free write percent, page 11-75
N/A	identity grab size, page 11-77
N/A	lock promotion HWM, page 11-80
N/A	lock promotion LWM, page 11-82
N/A	lock promotion PCT, page 11-83
N/A	max engine freelocks, page 11-39
N/A	o/s async i/o enabled, page 11-60
N/A	o/s file descriptors, page 11-61
N/A	page lock spinlock ratio, page 11-42
N/A	page utilization percent, page 11-29
N/A	partition groups, page 11-92
N/A	partition spinlock ratio, page 11-93
N/A	size of auto identity column, page 11-94
N/A	table lock spinlock ratio, page 11-44
N/A	total data cache size, page 11-25

Name In Previous Release	New Name
N/A	user log cache size, page 11-111
N/A	user log cache spinlock ratio, page 11-112

What Are Configuration Parameters?

Configuration parameters are user-definable settings that control various aspects of SQL Server's behavior. SQL Server supplies default values for all configuration parameters; these defaults are reasonable, given the following assumptions:

- At least 15MB of RAM is dedicated to SQL Server
- SQL Server is subject to a large volume of update activity
- A small number of stored procedures are used frequently

If your application differs significantly from these assumptions, read this chapter carefully to determine which configuration parameters you should reset to optimize server performance. Also, see the *Performance and Tuning Guide* for further information on using **sp_configure** to tune SQL Server.

How to Modify Configuration Parameters

You set or change configuration parameters in one of two ways:

- By executing the system procedure **sp_configure** with the appropriate parameters and values
- By hand-editing your configuration file and then invoking **sp_configure** with the **configuration file** option

Configuration parameters are either **dynamic** or **static**. Dynamic parameters go into effect as soon as you execute **sp_configure**. Static parameters require SQL Server to reallocate memory, and they take effect only after SQL Server has been restarted. The description of each parameter indicates whether it is static or dynamic.

Who Can Modify Configuration Parameters

The roles required for using **sp_configure** are:

- Any user can execute **sp_configure** to display information about parameters and their current values.

- Only System Administrators and System Security Officers can execute **sp_configure** to modify configuration parameters.
- Only System Security Officers can execute **sp_configure** to modify values for:

  allow updates
  audit queue size
  remote access
  systemwide password expiration

Using *sp_configure*

The system procedure **sp_configure** displays and resets configuration parameters. You can restrict the number of parameters **sp_configure** displays by using **sp_displaylevel** to set your display level to one of three values:

- Basic
- Intermediate
- Comprehensive

For information about display levels, see "User-Defined Subsets of the Parameter Hierarchy: Display Levels" on page 11-12. For information about **sp_displaylevel**, see **sp_displaylevel** in the *SQL Server Reference Manual*.

Each parameter belongs to a group based on the area of server behavior it affects. See "The Parameter Hierarchy" on page 11-11 for more information.

The following table describes the syntax for **sp_configure**. The information in the "Effect" column assumes that your display level is set to "comprehensive."

Table 11-1: sp_configure syntax

Command	Effect
sp_configure	Displays all configuration parameters by group, their current values, their default values, the value to which they have most recently been set, and the amount of memory this particular setting uses.

Table 11-1: sp_configure syntax (continued)

Command	Effect
sp_configure "*parameter*"	Displays current value, default value, most recently changed value, and amount of memory used by setting for all parameters matching parameter.
sp_configure "*parameter*", *value*	Resets *parameter* to *value*.
sp_configure "*parameter*", *0*, "*default*"	Resets parameter to its default value.
sp_configure "***group_name***"	Displays all configuration parameters in *group_name*, their current values, their default values, the value to which they have most recently been set, and the amount of memory this particular setting uses.
sp_configure "***configuration file***", ***0***, "***sub_command***", "***file_name***"	Sets configuration parameters from the configuration file. See "Using sp_configure with a Configuration File" on page 11-7 for descriptions of subcommands and their effects.

Syntax Elements

In Table 11-1 the following elements are used:

- *parameter* is any valid SQL Server configuration parameter or parameter substring.
- *value* is any integer within the valid range for that parameter. (See the descriptions of individual parameters for valid range information.) Parameters that are toggles have only two valid values: 1 (on) and 0 (off).
- *group_name* is the name of any group in the parameter hierarchy.

Parameter Parsing

sp_configure parses parameters (and parameter name fragments) as "%parameter%". A string that does not uniquely identify a particular parameter will return values for all the parameters matching the string. For example:

```
sp_configure "lock"
```

returns values for **lock shared memory into physical memory, number of locks for all users, lock promotion HWM, lock promotion LWM, lock promotion PCT, print deadlock information**, and **deadlock retries**.

➤ ***Note***

If you attempt to set a parameter value with a non-unique parameter name fragment, **sp_configure** returns the current values for all parameters matching the fragment and asks for a unique parameter name.

Using *sp_configure* with a Configuration File

SQL Server configuration can be done either interactively, by using **sp_configure** as described above, or noninteractively, by invoking **sp_configure** with a configuration file. Configuration files can be used for several reasons:

- You can replicate a specific configuration across multiple servers by using the same configuration file.
- You can use a configuration file as a baseline for testing configuration values on your server.
- You can use a configuration file to do validation checking on parameter values before actually setting the values.
- You can create multiple configuration files and switch between them as your resource needs change.

Each time you modify a configuration parameter with **sp_configure**, a new configuration file is created, using the naming convention *server_name.001, server_name.002, server_name.003 ... server_name.999*.

If you rename a configuration file, and you keep the *server_name* part of the file name, be sure to include at least one alphabetic character in the extension. Alternatively, you can change the *server_name* part of the file name. Doing this will avoid confusion with the configuration files that SQL Server automatically generates when you modify a parameter.

The syntax for using the **configuration file** option with **sp_configure** is:

```
sp_configure "configuration file", 0, "subcommand",
   "file_name"
```

where:

- "*configuration file*" (include quotes) specifies the configuration file parameter.

- 0 must be included as the second parameter to **sp_configure** for backward compatibility.
- *file_name* specifies the configuration file you want to use in conjunction with any *subcommand*.

Optional Subcommands

The four subcommands described below can be used with configuration files.

write

write creates *file_name* from the current configuration. If you do not specify a directory with *file_name*, the file is created in *$SYBASE*. If *file_name* already exists, a message is written to the error log; the existing file is renamed using the convention *file_name.001*, *file_name.002*, and so on. If you have changed a static parameter but have not restarted your server, **write** gives you the **currently running value** for that parameter.

read

read performs validation checking on values contained in *file_name* and reads those values that pass validation into the server. If any parameters are missing from *file_name*, the current values for those parameters are used.

If the value of a static parameter in *file_name* is different from its current running value, **read** fails and a message is printed. However, validation is still performed on the values in *file_name*.

verify

verify performs validation checking on the values in *file_name*. This is useful, as it prevents you from attempting to configure your server with invalid configuration values.

restore

restore creates *file_name* with the values in *sysconfigures*. This is useful if all copies of the configuration file have been lost and you need to generate a new copy. If you do not specify a directory with *file_name*, the file is created in *$SYBASE*.

Examples

```
sp_configure "configuration file", 0, "read",
"$SYBASE/srv.config"
```

The above example performs validation checking on the values in the file *$SYBASE/srv.config* and reads the parameters that pass validation into the server. Current run values are substituted for values that do not pass validation checking.

```
sp_configure "configuration file", 0, "write",
"$SYBASE/my_server.config"
```

The above example creates the file *$SYBASE/my_server.config* and writes the current configuration values the server is using to that file.

```
sp_configure "configuration file", 0, "verify",
"$SYBASE/generic.config"
```

The above example runs validation checking on the values in the file *~sybase/generic.config*.

Editing the Configuration File by Hand

The configuration file is an operating system ASCII file that can be edited with any text editor that can save files in ASCII format. The syntax for each parameter is:

```
parameter_name={value | default}
```

where *parameter_name* is the name of the parameter you want to specify; *value* is the numeric value you want to set *parameter_name* to; “default” specifies that you want to use the default value for *parameter_name*. If you specify “default,” do not include *value*.

Examples:

```
number of extent io buffers=100
```

sets the parameter **number of extent i/o buffers** to 100.

```
cpu accounting flush interval=default
```

specifies that the default value for the parameter **cpu accounting flush interval** should be used. Note that because “default” was specified, no value was given.

➤ ***Note***

You cannot edit the configuration file that is currently in use by a running SQL Server. However, you can make a copy of it either by using the **write** option to **sp_configure** or by using your operating system's file copy command. You can then edit the copy.

When you hand-edit a configuration file, be aware that your edits will not be validated until you either check the file using the **verify** option or restart SQL Server with that configuration file.

If all your configuration files are lost or corrupted, you can re-create one from a running server by using the **restore** subcommand and specifying a name for the new file. The parameters in the new file will be set to the values with which your server is currently running.

Permissions for Configuration Files

Configuration files are nonencrypted ASCII text files. By default, they are created with read and write permissions set for the file owner and read permission set for all other users. (If you created the configuration file at the operating system level, you are the file owner; if you created the configuration file from SQL Server using the **write** or **restore** subcommand, the file owner is the user who booted SQL Server. Normally, this is the user "sybase.") To restrict access to configuration files, use your operating system's file permission command to set read, write, and execute permissions as appropriate.

➤ ***Note***

You need to set permissions accordingly on **each** configuration file created.

Backing Up Configuration Files

Configuration files are not automatically backed up when you back up your master database. They are operating system files, and you should back them up in the same way you back up your other operating system files.

Starting SQL Server with a Configuration File

SQL Server can be invoked with a configuration file by using the following command:

```
dataserver -c configuration_file
```

where *configuration_file* is the path name of the configuration file you want to use. If the configuration file you specify does not exist, SQL Server will not boot.

If the command is successful, the file *$SYBASE/$DSLISTEN.bak* is created. This file contains the configuration values stored in *sysconfigures* prior to the time *sysconfigures* was updated with the values read in from the configuration file you specified. This file is overwritten with each subsequent start-up.

The Parameter Hierarchy

Configuration parameters are grouped according to the area of SQL Server behavior they affect. This makes it easier to identify all parameters that might need to be tuned in order to improve a particular area of SQL Server performance.

The groups are:

- Backup / Recovery
- Cache Manager
- Disk I / O
- General Information
- Languages
- Lock Manager
- Memory Use
- Network Communications
- Operating System Resources
- Physical Memory
- Processors
- SQL Server Administration
- User Environment

While each parameter has a primary group to which it belongs, many have secondary groups to which they also belong. For instance, the parameter **number of remote connections** belongs primarily in the Network Communications group, but also secondarily in the SQL Server Administration group and the Memory Use group. This reflects the fact that some parameters have implications for a number of areas of SQL Server behavior. **sp_configure** displays parameters in all the groups to which they belong.

The syntax for displaying all groups and their associated parameters (and the current values for the parameters) is:

sp_configure

➤ ***Note***

The number of parameters **sp_configure** returns depends on the value to which you have your display level set. If you have your display level set to "basic,*"* **sp_configure** won't return parameters that are designated as either "intermediate" or "comprehensive." If you have your display level set to "intermediate," **sp_configure** returns those parameters designated as "basic" and "intermediate," but not those designated as "comprehensive." See "User-Defined Subsets of the Parameter Hierarchy: Display Levels" on page 11-12 for further information about display levels.

The syntax for displaying a particular group and its associated parameter is:

sp_configure *"group_name"*

where *group_name* is the name of the group you are interested in. For example, to display the Disk I/O group, type:

sp_configure "Disk I/O"

```
Group: Disk I/O

Parameter Name           Default Memory Used Config Value Run Value
--------------           ------- ----------- ------------  --------
allow sql server async i/o     1           0            1         1
disk i/o structures          256           0          256       256
number of devices             10           0           10        10
page utilization percent      95           0           95        95
```

User-Defined Subsets of the Parameter Hierarchy: Display Levels

Depending on your use of SQL Server, you may need to adjust some parameters more frequently than others. You may find it is easier to work with a subset of parameters than having to see the entire group when you are working with only a few. You can set your display level to one of three values to give you the subset of parameters that best suits your working style.

The default display level is "comprehensive." When you set your display level, the setting persists across multiple sessions. However, you can reset it at any time to see more or fewer configuration parameters.

- “Basic” level shows just the most basic parameters. It is appropriate for very general server tuning.
- “Intermediate” level shows you parameters that are somewhat more complex, as well as showing you all the “basic” parameters. This level is appropriate for a moderately complex level of server tuning.
- “Comprehensive” level shows you all the parameters, including the most complex ones. This level is appropriate for users doing highly detailed server tuning.

The syntax for showing your current display level is:

```
sp_displaylevel
```

The syntax for setting your display level is:

```
sp_displaylevel "user_name" [,"basic" |
"intermediate" | "comprehensive"]
```

where *user_name* is your SQL Server login name.

The Effect of the Display Level on *sp_configure* Output

If your display level is set to either “basic” or “intermediate.” **sp_configure** returns only a subset of the parameters that are returned when your display level is set to “comprehensive.” For instance, if you have your display level set to “intermediate,” and you want to see the parameters in the Disk I/O group, type:

```
sp_configure "Disk I/O"
```

The output would look like:

```
Group: Disk I/O

Parameter Name          Default Memory Used Config Value Run Value
--------------          ------- ----------- ------------  --------
allow sql server async i/o     1           0            1         1
disk i/o structures          256           0          256       256
number of devices             10           0           10        10
page utilization percent      95           0           95        95
```

However, this is only a subset of the parameters in the Languages group, because one parameter in that group (**default sortorder id**) is defined as Comprehensive level.

The *reconfigure* Command

Previous releases of SQL Server required you to execute **reconfigure** after executing **sp_configure**. This is no longer required. The **reconfigure** command still exists, but it no longer has any effect. It is included in this release only in order to allow pre-11.0 scripts to run without modification.

➤ ***Note***

If you have scripts that include **reconfigure**, you should change them at your earliest convenience. Although **reconfigure** is included in this release, it may not be supported in subsequent releases.

Performance Tuning with *sp_configure* and *sp_sysmon*

sp_sysmon is a system procedure that monitors SQL Server performance and generates statistical output describing the behavior of your SQL Server system. See Chapter 19, “Monitoring SQL Server Performance with sp_sysmon”, in the *Performance and Tuning Guide* for information about how to use **sp_sysmon**.

You can run **sp_sysmon** before and after using **sp_configure** to adjust configuration parameters. The output gives you a basis for performance tuning and lets you observe the results of configuration changes.

This chapter gives cross-references to the *Performance and Tuning Guide* for **sp_configure** parameters that can affect SQL Server performance.

Output from *sp_configure*

The sample output below shows the kind of information **sp_configure** prints if you have your display level set to “comprehensive” and you execute it with no parameters. The values it prints will vary, depending on your platform and on what values you have already changed.

```
1> sp_configure

Group: General Information
Parameter Name         Default Memory Used Config Value Run Value
--------------         ------- ----------- ------------ ---------
configuration file           0          0           0 /remote/pub

Group: Backup/Recovery

Parameter Name         Default Memory Used Config Value Run Value
--------------         ------- ----------- ------------ ---------
recovery interval in minutes 5          0            5        5
tape retention in days       0          0            0        0
recovery flags               0          0            0        0
...
```

➤ ***Note***

All other configuration groups and parameters will appear in output (if you have your display level set to Comprehensive).

The “Default” column displays the value SQL Server is shipped with. If you do not explicitly reconfigure a parameter, it retains its default value.

The “Memory Used” column displays the amount of memory used (in kilobytes) by the parameter at its current value. Some related parameters draw from the same memory pool. For instance, the memory used for **stack size** and **stack guard size** is already accounted for in the memory used for **number of user connections**. If you added the memory used by each of these parameters separately, it would total more than the amount actually used. In the “Memory Used” column, parameters that “share” memory with other parameters are marked with a hash mark (“#”).

The “Config Value” column displays the most recent value to which the configuration parameter has been set with **sp_configure**. When you execute **sp_configure** to modify a dynamic parameter:

- The configuration and run values are updated
- The configuration file is updated
- The change takes effect immediately

When you modify a static parameter:

- The configuration value is updated

- The configuration file is updated
- The change takes effect only when you restart SQL Server

The "Run Value" column displays the value SQL Server is currently using. It changes when you modify a dynamic parameter's value with **sp_configure** and, for static parameters, after you restart SQL Server.

The *sysconfigures* and *syscurconfigs* Tables

The report displayed by **sp_configure** is constructed from a table called *spt_values*. The *spt_values* table is a "look-up table" in the *master* database that stores records that refer to locks, permissions, configuration parameters, and other system information.

The *master..sysconfigures* and *master..syscurconfigs* system tables store configuration parameters. The *value* field of each table contains the value that has been set by executing **sp_configure**.

The *status* field of *sysconfigures* cannot be updated by the user. Status 1 means "dynamic," indicating that new values for these configuration parameters take effect immediately when **sp_configure** is executed. Status 0 indicates a parameter is "static" and takes effect only after **sp_configure** is executed **and** SQL Server is 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 restarted SQL Server (to modify static parameters), or
- The default for many values in *sysconfigures* is 0, which indicates that SQL Server calculates default values for that parameter.

The run values are stored in the system table *syscurconfigs*. This table is created dynamically when it is queried. **sp_configure** performs a join on *sysconfigures* and *syscurconfigs* to display the values that are currently in use—the **run values**—and the configured values.

How SQL Server Uses Memory

Many configuration parameters consume memory; this memory is allocated from the amount you configure for SQL Server. In general, you want to configure as much memory for SQL Server as you have physically available. If you have specified more memory than is available, SQL Server may not be able to boot, or if it does boot, the operating system page fault rate may increase significantly. On the other hand, if you specify too little memory, SQL Server may not boot; if it does boot, it may run very slowly due to frequent disk I/O.

To learn more about how SQL Server uses memory, and how to determine the memory available on your system, see Chapter 8, “Configuring Memory.” Also see the parameter descriptions in this chapter to understand the ways in which individual parameters consume memory.

Details on Configuration Parameters

The following sections give both summary and detailed information about each of the configuration parameters. Parameters are listed by group; within each group they are listed alphabetically.

Backup and Recovery

The following parameters configure SQL Server for backing up and recovering data.

print recovery information

Summary Information	
Name in previous release	**recovery flags**
Default value	0 (off)
Valid values	0 (off), 1 (on)
Status	Static
Display level	Intermediate
Required role	System Administrator

The **print recovery information** parameter sets a toggle that determines what information SQL Server displays on the console during recovery. (Recovery is done on each database at SQL Server start-up, and when a database dump is loaded.) The default 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 processed during recovery, including whether it was aborted or committed.

recovery interval in minutes

Summary Information	
Name in previous release	**recovery interval**
Default value	5
Range of values	1–32767
Status	Dynamic
Display level	Basic
Required role	System Administrator

The **recovery interval in minutes** parameter sets the maximum number of minutes per database that SQL Server should use to complete its recovery procedures in case of a system failure. The recovery procedure rolls transactions backwards or forwards starting from the transaction that the checkpoint process indicates as the oldest active transaction. The recovery process has more or less work to do depending on the value of **recovery interval in minutes.**

SQL Server estimates that 6000 rows in the transaction log require 1 minute of recovery time. However, different types of log records can take more or less time to recover. If you set **recovery interval in minutes** to 3, the checkpoint process writes changed pages to disk only when *syslogs* contains more than 18,000 rows.

➤ ***Note***

The recovery interval has no effect on long-running, minimally logged transactions (such as **create index**) 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. To avoid lengthy delays, dump each database immediately after creating an index on one of its tables.

SQL Server uses this **recovery interval in minutes** 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 in cache that have been modified) to disk. The checkpoint also performs a few other maintenance tasks, including truncating the transaction log for each database for which the **truncate log on chkpt** option has been set. About once per minute, the sleeping checkpoint process "wakes up," checks the **truncate log on chkpt** setting, and checks the recovery interval to determine if a checkpoint is needed. The following illustration shows the logic used by SQL Server during this process.

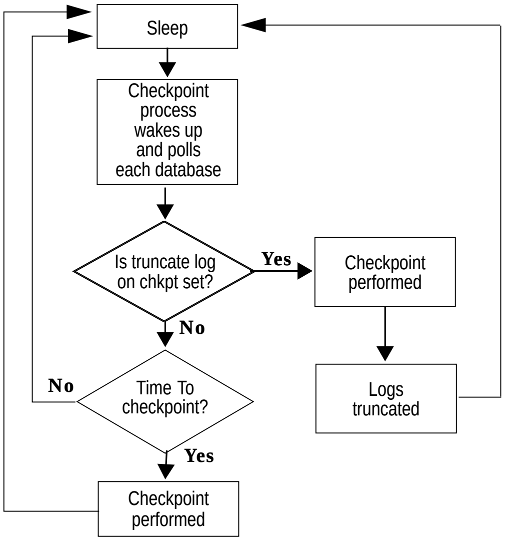


Figure 11-1: The checkpoint process

You may want to change the recovery interval if your application and its use 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, or spikes, the system very slightly. On the other hand, setting the recovery interval too high might cause the time to recover to be unacceptably long. (The spikes caused by checkpointing can be reduced by reconfiguring the **housekeeper free write percent** parameter. See "housekeeper free write percent" on page 11-75 for further information.) For more information on the performance implications of **recovery interval in minutes**, see "Speed of Recovery" on page 15-34 in the *Performance and Tuning Guide.*

Use **sp_sysmon** to determine how a particular recovery interval affects the system. See "Recovery Management" on page 19-63 in the *Performance and Tuning Guide*.

tape retention in days

Summary Information	
Name in previous release	**tape retention**
Default value	0
Range of values	0–365
Status	Static
Display level	Intermediate
Required role	System Administrator

The **tape retention in days** parameter specifies the number of days you intend to retain each tape after it has been used for either a database or transaction log dump. It is intended to keep you from accidentally overwriting a dump tape.

For example, if you have set **tape retention in days** to 7 days, and you try to use the tape before 7 days have elapsed since the last time you dumped to that tape, Backup Server issues a warning message to the tape operator.

You can override the warning by using the **with init** option when executing the dump command. Be aware that doing this will cause the tape to be overwritten and all data on the tape to be lost.

Both the **dump database** and **dump transaction** commands provide a **retaindays** option, which overrides the **tape retention in days** value for a particular dump. See "Protecting Dump Files from Being Overwritten" on page 19-22 for more information.

Cache Manager

The parameters in this group configure the data and procedure caches.

memory alignment boundary

Summary Information	
Name in previous release	**calignment**
Default value	2048
Range of values	2048–16384
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **memory alignment boundary** parameter determines on which boundary data caches are aligned.

Some machines perform I/O more efficiently when structures are aligned on a particular memory address boundary. In order to preserve this alignment, values for **memory alignment boundary** should always be multiples of 2K.

➤ ***Note***

The **memory alignment boundary** parameter is included for support of certain hardware platforms. Do not modify it unless you are instructed to do so by Sybase Technical Support.

number of index trips

Summary Information	
Name in previous release	**cindextrips**
Default value	0
Range of values	0–65535
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **number of index trips** parameter specifies the number of times an aged index page traverses the most-recently-used / least-recently-used (MRU / LRU) chain before it is considered for swapping out. As you increase the value of **number of index trips**, the greater amount of time old index pages stay in cache.

A data cache is implemented as an MRU / LRU chain. As the user threads access data and index pages, these pages are placed on the MRU end of the cache's MRU / LRU chain. In some high transaction environments (and in some benchmarks), it is desirable to keep index pages in cache, since they will probably be needed again soon. Setting **number of index trips** higher keeps index pages in cache longer; setting it lower allows index pages to be swapped out of cache sooner.

➤ ***Note***

If the cache used by an index is relatively small (especially if it shares space with other objects) and you have a high transaction volume, you should be careful not to set **number of index trips** too high. The cache will flood with pages that do not age out, and this may lead to processes timing out while waiting for cache space.

number of oam trips

Summary Information	
Name in previous release	**coamtrips**
Default value	0
Range of values	0– 65535
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **number of oam trips** parameter specifies the number of times an aged OAM page traverses the MRU / LRU chain before it is considered for swapping out. As you increase the value of **number of oam trips** the longer aged OAM pages stay in cache.

Each table, and each index on a table, has an **Object Allocation Map** (**OAM**) page. The OAM page holds information on pages allocated to the table or index, and is checked when a new page is needed for the index or table. (See “page utilization percent” on page 11-29 for further information.) A single OAM page can hold allocation mapping for between 2000 and 63,750 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 some environments and benchmarks that involve significant allocations of space (that is, massive bulk copy operations), keeping OAM pages in cache longer improves performance. Setting **number of oam trips** higher keeps OAM pages in cache.

➤ ***Note***

If the cache is relatively small and used by a large number of objects, you should not set **number of oam trips** too high. This will result in the cache being flooded with OAM pages that do not age out, and user threads will begin to time out.

procedure cache percent

Summary Information	
Name in previous release	**procedure cache**
Default value	20
Range of values	1–99
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **procedure cache percent** parameter specifies the amount 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 value to which **procedure cache percent** has been set.

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 does not need to read it from the disk. SQL Server also uses space in the procedure cache to compile queries 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 is not necessary to read it from the disk. The data cache holds recently used data and index pages; older pages are flushed to disk as space is needed for new pages.

Both caches are managed in MRU / LRU (most-recently-used / least-recently-used) fashion.

Since the optimum value for **procedure cache percent** 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 parameter to 50 during your development cycle and resetting it to 20 when your application becomes stable. For further information on configuring procedure caches, see “The Procedure Cache” on page 15-4 of the *Performance and Tuning Guide*.

total data cache size

Summary Information	
Name in previous release	N/A
Default value	N/A
Range of values	N/A
Status	Calculated
Display level	Basic
Required role	System Administrator

The **total data cache size** parameter represents the amount of memory that is currently available for data, index, and log pages. It is a calculated value that is not directly user-configurable.

The amount of memory available for the data cache can be affected by a number of factors, including:

- The amount of physical memory available on your machine
- The values to which the following parameters are set:
  - **total memory**
  - **number of user connections**
  - **total procedure cache percent**
  - **number of open databases**
  - **number of devices**
  - **number of open database objects**

A number of other parameters also affect the amount of available memory, but to a lesser extent.

For information on how SQL Server allocates memory and for information on data caches, see “How SQL Server Uses Memory” on page 11-16.

Disk I/O

The parameters in this group configure SQL Server's disk I/O.

allow sql server async i/o

Summary Information	
Name in previous release	**T1603** (trace flag)
Default value	1 (on)
Valid values	0 (off), 1 (on)
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **allow sql server async i/o** parameter is a toggle that enables SQL Server to run with asynchronous disk I/O. In order to use asynchronous disk I/O, you have to enable it on **both** SQL Server **and** your operating system. (See "o/s async i/o enabled" on page 11-60 for information on determining whether or not asynchronous disk I/O is currently enabled at the operating system level. See your operating system documentation for information on enabling asynchronous I/O at the OS level.)

In virtually all circumstances, disk I/O runs faster asynchronously than synchronously. This is because when SQL Server issues an asynchronous I/O, it does not have to wait for a response before issuing further I/Os.

If you have SQL Server configured to use block I/O and have your master device on an operating system file, asynchronous I/O may be slightly slower than synchronous I/O. However, doing this is not advised, because SQL Server cannot guarantee full data recovery in the event of a system crash if your master device is on an operating system file.

disk i/o structures

Summary Information	
Name in previous release	**cnblkio**
Default value	256
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **disk i/o structures** parameter specifies the initial number of disk I/O control blocks SQL Server allocates on start-up.

User processes require a disk I/O control block before SQL Server can initiate an I/O request for the process. The memory for disk I/O control blocks is preallocated when SQL Server starts. You should configure **disk i/o structures** as high as your operating system allows, to minimize the chance of running out of disk I/O structures. For example, if your operating system allows *N* concurrent disk I/Os, configuring **disk i/o structures** to *N* would require:

```
N * bytes/structure = amount of memory
```

Refer to your operating system documentation for information on concurrent disk I/Os.

Use **sp_sysmon** to determine if you need to allocate more disk I/O structures. See “Disk I/O Structures” in “Sample Output for Disk I/O Management” on page 19-67 in the *Performance and Tuning Guide*.

number of devices

Summary Information	
Name in previous release	**devices**
Default value	10
Range of values	1–256
Status	Static
Display level	Basic
Required role	System Administrator

The **number of devices** parameter controls the number of database devices SQL Server can use. It does not include devices used for database or transaction log dumps. Each device uses less than 512 bytes of memory.

Each device number must be unique among the device numbers used by SQL Server. The number 0 is reserved for the master device. Legal numbers are 4–256, inclusive. However, the highest number must be one less than the number of database devices you have configured for SQL Server. For example, if you configured your server for 10 devices, the legal range of device numbers is 1–9, inclusive.

To determine which numbers are currently in use, run **sp_helpdevice** and look in the *device_number* column of output.

If you drop a device with **sp_dropdevice**, you cannot reuse its **vdevno** until you reboot SQL Server.

If you want to lower the **number of 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 the highest value in use:

```
select max(low/power(2,24))+1
    from master..sysdevices
```

◆ ***WARNING!***

If you set number of devices too low, SQL Server will not be able to recover databases that use devices with higher numbers.

page utilization percent

Summary Information	
Name in previous release	N/A
Default value	95
Range of values	1–100
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The page utilization percent parameter is used during page allocations to control whether SQL Server scans a table's OAM (**Object Allocation Map**) to find unused pages, or simply allocates a new extent to the table. (See "number of oam trips" on page 11-23 for more information on the OAM.)

If page utilization percent is set to 100, SQL Server always scans through all OAM pages to find unused pages allocated to the object before allocating a new extent. When this parameter is set lower than 100, SQL Server compares the page utilization percent setting to the ratio of used and unused pages allocated to the table, as follows:

```
100 * used pages/(used pages + unused pages)
```

If the page utilization percent setting is higher than the ratio, SQL Server allocates a new extent rather than searching for the unused pages.

For example, when inserting data into a 10GB table that has 120 OAM pages and only one unused data page:

- A page utilization percent of 100 tells SQL Server to scan through all 120 OAM pages to locate the unused data page.
- A page utilization percent of 95 allows SQL Server to allocate a new extent to the object, because 95 is lower than the ratio of used pages to used and unused pages.

A low page utilization percent value results in more unused pages. A high page utilization percent value slows page allocations in large tables, as SQL Server performs an OAM scan to locate each unused page before allocating a new extent.

If page allocations (especially in the case of large inserts) seem to be slow, you can lower the **page utilization percent**, but reset it after inserting all your data, as a lower setting results in unused pages in all tables.

Bulk copy ignores the **page utilization percent** setting and always allocates new extents until there are no more extents available in the database.

General Information

The parameters in this group are not related to a particular area of SQL Server behavior.

configuration file

Summary Information	
Name in previous release	N/A
Default value	0
Range of values	N/A
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **configuration file** parameter specifies the location of the configuration file currently in use. See “Using sp_configure with a Configuration File” on page 11-7 for a complete description of configuration files.

Note that in **sp_configure** output the “Run Value” column displays only ten characters. For this reason the output may not display the entire path and name of your configuration file.

Languages

The parameters in this group configure languages, sort orders, and character sets.

default character set id

Summary Information	
Name in previous release	**default character set id**
Default value	1
Range of values	0–255
Status	Static
Display level	Intermediate
Required role	System Administrator

The **default character set id** parameter 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 12, "Configuring Character Sets, Sort Orders, and Message Language," for a discussion of how to change character sets and sort orders.

default language id

Summary Information	
Name in previous release	**default language**
Default value	0
Range of values	0–32767
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **default language id** parameter 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.

default sortorder id

Summary Information	
Name in previous release	**default sortorder id**
Default value	50
Range of values	0–255
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **default sortorder id** parameter 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 12, “Configuring Character Sets, Sort Orders, and Message Language,” for further information.

number of languages in cache

Summary Information	
Name in previous release	**language in cache**
Default value	3
Range of values	3–100
Status	Static
Display level	Intermediate
Required role	System Administrator

The **number of languages in cache** parameter indicates the maximum number of languages that can be simultaneously held in the language cache. The default is 3.

Lock Manager

The parameters in this group configure locks.

address lock spinlock ratio

Summary Information	
Name in previous release	N/A
Default value	100
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

For SQL Servers running with multiple engines, you can set the ratio of **spinlocks** protecting the address locks hash table using the **address lock spinlock ratio** configuration parameter. A SQL Server configured with only one engine (defined by the **max online engines** parameter) has only one spinlock, regardless of the value specified for **address lock spinlock ratio**, since only that one engine will access the hash table. SQL Server SMP systems with two or more engines can have a maximum of 1031 spinlocks protecting the table.

SQL Server manages the acquiring and releasing of address locks using an internal hash table with 1031 rows (known as hash buckets). This table can use one or more spinlocks to serialize access between processes running on different engines. A spinlock is a simple internal locking mechanism which prevents a process from accessing the resource currently used by another process. All processes trying to access the resource must wait (or “spin”) until the lock is released.

Figure 11-2 illustrates the relationship between the spinlocks and the internal address locks hash table. By default, SQL Server SMP systems with more than one engine are configured with 11 spinlocks for the table. The first 10 spinlocks protect 100 rows each, and the eleventh spinlock protects the remaining 31 rows.

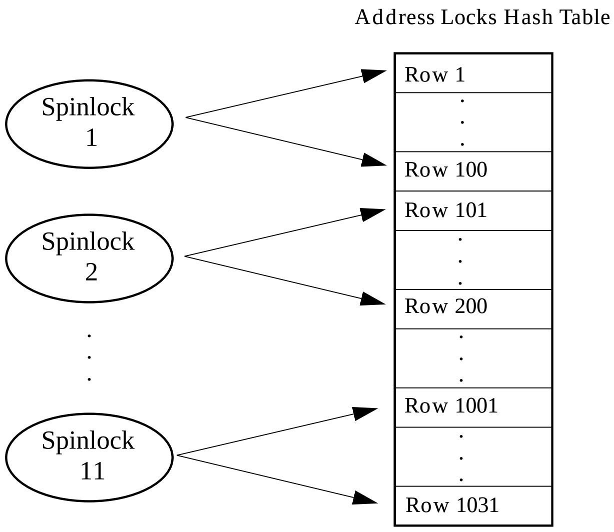


Figure 11-2: Relationship between spinlocks and address locks hash table

The **address lock spinlock ratio** parameter specifies the number of rows in the address locks hash table protected by one spinlock (rows per spinlock), not the total number of spinlocks. So, the total number of spinlocks protecting the table is expressed by the following formula:

(1031 / **address lock spinlock ratio** value) + 1 for any remainder

For example, if you specify a ratio of 50 rows per spinlock, the total number of spinlocks equals 21 (1031 / 50 plus 1); 20 spinlocks protect 50 rows each while the twenty-first spinlock protects the remaining 31 rows. You can define that ratio as follows:

```
sp_configure "address lock spinlock ratio", 50
```

SQL Server's default value for **address lock spinlock ratio** is 100, which defines 11 spinlocks for the address locks hash table. If you specify a value of 1031 or greater for **address lock spinlock ratio**, SQL Server uses only one spinlock for the entire table. **address lock spinlock ratio** is **not** a dynamic configuration parameter. You must shut down and restart SQL Server for any change to take effect.

In theory, protecting one row with one spinlock (by defining an **address lock spinlock ratio** of 1 for a total of 1031 spinlocks) would provide the least contention for a spinlock and the highest possible number of simultaneous address lock operations. However, benchmark studies show the contention for the spinlocks is generally less than 3 percent using SQL Server's default of 11 spinlocks (an **address lock spinlock ratio** of 100).

Use **sp_sysmon** to measure address spinlock contention. See "Address Locks" on page 19-43 in the *Performance and Tuning Guide*. If **sp_sysmon** output indicates address lock contention greater than 3 percent, try decreasing the value of the **address lock spinlock ratio** parameter.

Each additional spinlock uses up to 256 bytes of memory. This additional memory requirement reduces the available memory for procedure and data caches, which may affect other performance aspects of your server. Therefore, you should be wary of making changes to this parameter.

deadlock checking period

Summary Information	
Name in previous release	N/A
Default value	500
Range of values	0– 2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **deadlock checking period** parameter specifies the minimum amount of time (in milliseconds) before SQL Server initiates a deadlock check for a process waiting on a lock to be released. Deadlock checking is time-consuming overhead for applications that wait without being deadlocked, and the overhead grows as the percentage of lock requests that must wait for a lock also increases.

If you set this value to a non-zero value *n*, SQL Server initiates a deadlock check after a process waits at least *n* ms. For example, you can make the processes wait at least 700 ms. on a lock before each deadlock check as follows:

```
sp_configure "deadlock checking period", 700
```

If you set this parameter to 0, SQL Server initiates the deadlock checking at the time each process begins to wait for a lock. It is a dynamic configuration value, so any change to it takes immediate effect. SQL Server's default value for **deadlock checking period** is 500 ms.

Configuring **deadlock checking period** to a higher value produces longer delays before deadlocks are detected. However, since SQL Server grants most lock requests before this time elapses, the deadlock checking overhead is avoided for those lock requests. So, if you expect your applications to deadlock infrequently, you can set **deadlock checking period** to a higher value and avoid the overhead of deadlock checking for most processes. Otherwise, the default value of 500 ms. should suffice. For more information about deadlock checking, see the *Performance and Tuning Guide*.

Use **sp_sysmon** to determine the frequency of deadlocks in your system and the best setting for the **deadlock checking period** parameter. See "Lock Management" on page 19-40 in the *Performance and Tuning Guide*.

deadlock retries

Summary Information	
Name in previous release	N/A
Default value	5
Range of values	0–2147483647
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **deadlock retries** parameter specifies the number of times a transaction retries to acquire a lock after it has become a deadlock victim. For instance, in Figure 11-3, Transaction A needs to acquire a lock on Table X, on which Transaction B currently holds an exclusive table lock. However, before Transaction B can release its exclusive table lock on Table X, it needs to acquire a lock on Table Y, on which Transaction A currently holds an exclusive table lock.

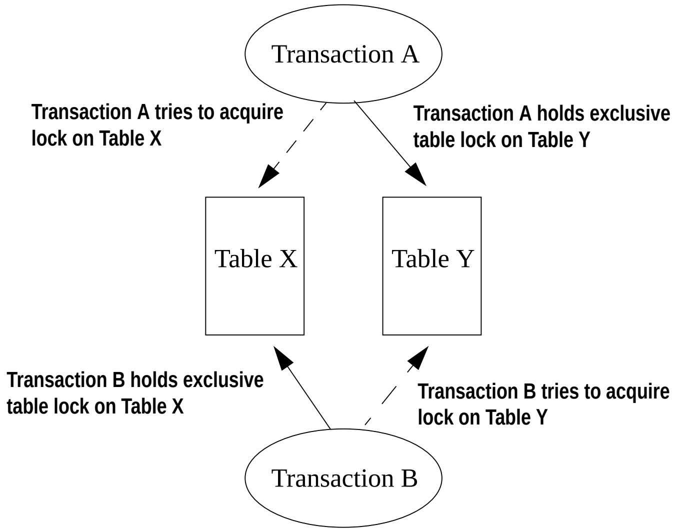

Transaction A
Transaction A tries to acquire lock on Table X
Transaction A holds exclusive table lock on Table Y
Table X
Table Y
Transaction B holds exclusive table lock on Table X
Transaction B tries to acquire lock on Table Y
Transaction B

Figure 11-3: Two transactions in a deadlock

Neither transaction can continue until the other releases its exclusive table lock. Transaction A has accumulated less CPU time than Transaction B, and thus becomes the deadlock victim. Transaction A continues trying to acquire a lock on Table X for as many times as the value of **deadlock retries**. If it fails to acquire the lock after that many retries, the transaction terminates.

freelock transfer block size

Summary Information	
Name in previous release	N/A
Default value	30
Range of values	1– 2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

When a process running on a multi-engine SQL Server requests a lock, it looks for one in its engine's freelock list. If the engine freelock list is out of locks, SQL Server moves a certain number of locks from its global freelock list to the engine freelock list. After a process completes, the locks released by those processes accumulate in the engine's freelock list. When the number of locks in the engine list reaches its maximum (defined by the **max engine freelocks** parameter), SQL Server moves a number of locks from the engine freelock list to the global freelock list. This replenishes the number of locks available to other engines from the global list.

The **freelock transfer block size** configuration parameter specifies the number of locks moved between the engine freelock lists and the global freelock list. SQL Server's default value for **freelock transfer block size** is 30, and the minimum allowed value is 1. For example, you can make 50 the transfer size as follows:

```
sp_configure "freelock transfer block size", 50
```

When you set this to a higher value, the frequency of transfers between the engine freelock lists and the global freelock list is reduced, which lowers the contention in accessing the global freelock list. However, a higher value can result in accessing many more lock structures than a process needs. The maximum value that SQL Server allows for **freelock transfer block size** cannot exceed more than half the maximum number of locks available to an engine's freelock list, as defined by the following formula:

((**max engine freelocks** percent value * **number of locks** value) /
max online engines value) / 2

For example, if **max engine freelocks** is set to 10 percent and the **number of locks** value is equal to 5000, the maximum value allowed for **freelock transfer block size** is 50 with a SQL Server running with five engines.

If you try to set it higher than its maximum value, SQL Server returns an error message and leaves **freelock transfer block size** unchanged. It also returns an error if the current **freelock transfer block size** value will exceed the maximum when you attempt to increase **max online engines** or decrease either **max engine freelocks** or **number of locks**. SQL Server sets this maximum to avoid draining the engine freelock lists of too many locks, which can force it to get more locks immediately from the global freelocks list.

max engine freelocks

Summary Information	
Name in previous release	N/A
Default value	10
Range of values	1–50
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

When a process running on a multi-engine SQL Server requests a lock, it looks for one in its engine's freelock list. If the engine freelock list is out of locks, SQL Server moves a certain number of locks (defined by the **freelock transfer block size** parameter) from its global freelock list to the engine freelock list. The total number of locks in the global freelock list is defined by the **number of locks** configuration parameter value. For single engine SQL Servers, the entire global freelock list is moved to the engine freelock list at server start-up time, regardless of the value of this parameter.

After an engine completes a process, all of locks held by that process are released and returned to that engine's freelock list. This reduces the contention of each engine accessing the global freelock list. However, if the number of locks released to the engine exceed the maximum number locks allowed in the engine's freelock list, SQL Server moves a number of locks (defined by **freelock transfer block size**) to the global freelock list. This replenishes the number of locks available to other engines from the global list.

You can specify the maximum number of locks available to the engine freelock lists as a percentage of the total **number of locks** available to your server with the **max engine freelocks** configuration parameter. You can specify any value from 1 percent to 50 percent. It is a dynamic configuration value, so any change to it takes immediate effect. For example, you can have 20 percent of the total number of locks be the maximum number of locks available to the engine freelock lists as follows:

```
sp_configure "max engine freelocks", 20
```

If your server has 5000 locks configured for the **number of locks** parameter, 20 percent (or 1000) of those locks represent the maximum number of locks available to the all the engine freelock lists. The actual maximum for each engine freelock list depends on the number of engines configured for SQL Server (**max online engines** parameter). If your server has five engines, the maximum for each engine freelock list is 1000 divided by 5, or 200 locks. Therefore, a change in the **number of locks** or **max online engines** configuration parameters can affect the maximum for each engine freelock list, even if the value for **max engine freelocks** remains the same.

If you set **max engine freelocks** too high for some servers, most of the available locks may end up in each engine’s freelock list, leaving very few locks in SQL Server’s global freelock list. If an engine freelock list becomes empty, it is also likely that the global freelock list is empty, which results in SQL Server error message 1279, even though other engines have locks in their freelock lists:

```
SQL Server has run out of locks on engine %d. Re-
run your command when there are fewer active
users, or contact a user with System Administrator
(SA) role to reconfigure max engine freelocks.
```

You should either decrease the configured value for **max engine freelocks** or increase the configured value of **number of locks** to avoid frequent occurrences of message 1279. This message differs from message 1204, which indicates that SQL Server has no more locks in the global freelock list and all engine freelock lists.

number of locks

Summary Information	
Name in previous release	**locks**
Default value	5000
Range of values	1000–2147483647
Status	Static
Display level	Basic
Required role	System Administrator

The **number of locks** parameter sets the total number of available locks for all users on SQL Server. Locks are not shared the way open databases and database objects are shared.

The number of locks in use at any one time varies significantly with the amount and type of activity on SQL Server. In order to see how many locks are in use at a particular time, use the stored procedure **sp_lock**.

You should start with the default value, 5000. If this proves insufficient, increase **number of locks** by another 1000. Continue doing this until you no longer run out of locks. (It is better to take this approach than to start by increasing **number of locks** to an arbitrarily high value. Each lock requires 72 bytes of memory. While this is a relatively small amount, it adds up. Consider that each 1000 locks consume 72K of memory.)

If you set **number of locks** too low and end up running out of locks, SQL Server displays two messages, one at the engine level and one at the server level.

page lock spinlock ratio

Summary Information	
Name in previous release	N / A
Default value	100
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

For SQL Servers running with multiple engines, you can set the ratio of spinlocks protecting the internal page locks hash table using the **page lock spinlock ratio** configuration parameter. A SQL Server configured with only one engine (defined by the **max online engines** parameter) has only one spinlock regardless of the value specified for **page lock spinlock ratio**, since only that one engine can access the hash table. SQL Server SMP systems with two or more engines can have a maximum of 1031 spinlocks protecting the table.

SQL Server manages the acquiring and releasing of page locks using an internal hash table with 1031 rows (known as hash buckets). This table can use one or more spinlocks to serialize access between processes running on different engines. A spinlock is a simple internal locking mechanism which prevents a process from accessing the resource currently used by another process. All processes trying to access the resource must wait (or “spin”) until the lock is released.

Figure 11-4 illustrates the relationship between the spinlocks and the internal page locks hash table. By default, SQL Server SMP systems with more than one engine are configured with 11 spinlocks for the table. The first 10 spinlocks protect 100 rows each; the eleventh spinlock protects the remaining 31 rows.

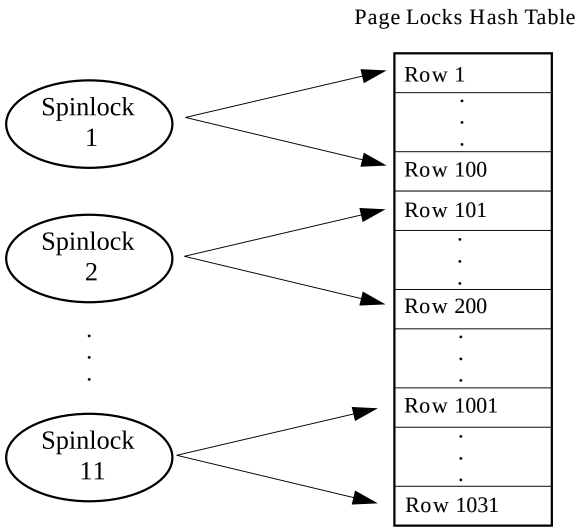


Figure 11-4: Relationship between spinlocks and page locks hash table

The **page lock spinlock ratio** parameter specifies the number of rows in the page locks hash table protected by one spinlock (rows per spinlock), not the total number of spinlocks. The higher the value you specify for **page lock spinlock ratio**, the lower the number of spinlocks protecting the table. Therefore, the total number of spinlocks protecting the table is expressed by the following formula:

(1031 / **page lock spinlock ratio** value) + 1 for any remainder

For example, if you specify a ratio of 50 rows per spinlock, the total number of spinlocks equals 21 (1031 divided by 50 plus 1); 20 spinlocks protect 50 rows each while the twenty-first spinlock protects the remaining 31 rows. You can define that ratio as follows:

```
sp_configure "page lock spinlock ratio", 50
```

SQL Server's default value for **page lock spinlock ratio** is 100, which defines 11 spinlocks for the page locks hash table. If you specify a value of 1031 or greater for **page lock spinlock ratio**, SQL Server uses only one spinlock for the entire table. **page lock spinlock ratio** is **not** a dynamic configuration parameter. You must shut down and restart SQL Server for any change to take effect.

In theory, protecting one row with one spinlock (by defining a **page lock spinlock ratio** of 1 for a total of 1031 spinlocks) would provide the least contention for a spinlock and the highest possible number of simultaneous page lock operations. However, benchmark studies show the contention for the spinlocks is generally less than 3 percent using SQL Server's default of 11 spinlocks (a **page lock spinlock ratio** of 100).

➤ ***Note***

Decreasing the value of the **page lock spinlock ratio** parameter should have little impact on the performance of SQL Server. The default setting for this parameter is correct for most servers.

Each additional spinlock uses up to 256 bytes of memory. This additional memory requirement reduces the available memory for procedure and data caches, which may affect other performance aspects of your server. Therefore, you should be wary of making changes to this parameter.

table lock spinlock ratio

Summary Information	
Name in previous release	N/A
Default value	20
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

For SQL Servers running with multiple engines, you can set the ratio of **spinlocks** protecting the internal table locks hash table using the **table lock spinlock ratio** configuration parameter. A SQL Server configured with only one engine (defined by the **max online engines** parameter) has only one spinlock, regardless of the value specified for **table lock spinlock ratio**, since only that one engine can access the hash table. SQL Server SMP systems with two or more engines can have a maximum of 101 spinlocks protecting the table.

SQL Server manages the acquiring and releasing of table locks using an internal hash table with 101 rows (known as hash buckets). This table can use one or more spinlocks to serialize access between processes running on different engines. A spinlock is a simple internal locking mechanism which prevents a process from accessing the resource currently used by another process. All processes trying to access the resource must wait (or “spin”) until the lock is released.

Figure 11-5 illustrates the relationship between the spinlocks and the internal table locks hash table. By default, SQL Server SMP systems with more than one engine are configured with 6 spinlocks for the table. The first 5 spinlocks protect 20 rows each; the sixth spinlock protects the last row.

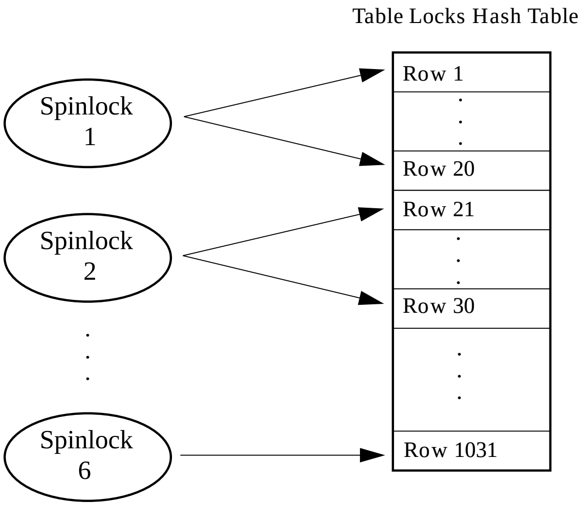


Figure 11-5: Relationship between spinlocks and table locks hash table

The **table lock spinlock ratio** parameter actually specifies the number of rows in the table locks hash table protected by one spinlock (rows per spinlock), not the total number of spinlocks. The higher the value you specify for **table lock spinlock ratio**, the lower the number of spinlocks protecting the table. So, the total number of spinlocks protecting the table is expressed by the following formula:

(101 / **table lock spinlock ratio** value) + 1 for any remainder

For example, if you specify a ratio of 8 rows per spinlock, the total number of spinlocks equals 13 (101 divided by 8 plus 1); 12 spinlocks protect 8 rows each; the thirteenth spinlock protects the remaining 6 rows. You can define that ratio as follows:

```
sp_configure "table lock spinlock ratio", 8
```

SQL Server's default value for **table lock spinlock ratio** is 20, which defines 6 spinlocks for the table locks hash table. If you specify a value of 101 or greater for **table lock spinlock ratio**, SQL Server uses only one spinlock for the entire table. **table lock spinlock ratio** is **not** a dynamic configuration parameter. You must shut down and restart SQL Server for any change to take effect.

In theory, protecting one row with one spinlock (by defining a **table lock spinlock ratio** of 1 for a total of 101 spinlocks) would provide the least contention for a spinlock and the highest possible number of simultaneous page lock operations. However, benchmark studies show the contention for the spinlocks is generally less than 3 percent using SQL Server's default of 6 spinlocks (a **table lock spinlock ratio** of 20).

➤ ***Note***

Decreasing the value of the **table lock spinlock ratio** parameter should have little impact on the performance of SQL Server. The default setting for this parameter is correct for most servers.

Each additional spinlock uses up to 256 bytes of memory. This additional memory requirement reduces the available memory for procedure and data caches, which may affect other performance aspects of your server. Therefore, you should be wary of making changes to this parameter.

Memory Use

The following parameters optimize SQL Server's memory use.

executable code size

Summary Information	
Name in previous release	**sql server code size**
Default value	N/A
Range of values	N/A
Status	Calculated
Display level	Basic
Required role	System Administrator

The **executable code size** parameter reports the size of the SQL Server executable. It is a calculated value and is not user-configurable.

Network Communication

Parameters in this group configure communication between SQL Server and remote servers, as well as with client programs.

allow remote access

Summary Information	
Name in previous release	**remote access**
Default value	1
Valid values	0 (off), 1 (on)
Status	Dynamic
Display level	Intermediate
Required role	System Security Officer

The **allow remote access** parameter 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 **allow remote access**.

Setting the value to 0 disables server-to-server remote procedure calls. Since SQL Server communicates with Backup Server via remote procedure calls, setting this parameter 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 parameters **number of remote logins**, **number of remote sites**, **number of remote connections**, and **remote server pre-read packets** are automatically set to their default values when **allow remote access** is enabled.

default network packet size

Summary Information	
Name in previous release	**default network packet size**
Default value	512
Range of values	512–524288
Status	Static
Display level	Intermediate
Required role	System Administrator

The **default network packet size** parameter 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 bytes. 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 **total memory** configuration parameter. This memory is allocated for network packets when SQL Server boots.

Each SQL Server user connection uses:

- One read buffer
- One read overflow buffer
- One write buffer

Each of these buffers requires **default network packet size** bytes. The total amount of memory allocated for network packets is:

number of user connections * 3 * **default network packet size**

When you provide a new value for this parameter, 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 **total memory** configuration and **number of user connections** parameters to be sure that you leave enough space for other SQL Server memory needs. When you restart SQL Server after changing the **default network packet size**, check the error log for messages that 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 **max network packet size** to at least the same size. Execute both **sp_configure** commands, and then restart your server.

If the value of **max network packet size** is greater than the value of **default network packet size**, you need to increase the value of **additional network memory.** See “additional network memory” on page 11-62 for further information.

Use **sp_sysmon** to see how changing the **default network packet size** parameter affects network I/O management and task switching. For example, try increasing **default network packet size** and check **sp_sysmon** output to see how this affects **bcp** for large batches. See “Network Packet Received” on page 19-20, “Network Packet Sent” on page 19-21, and “Network I/O Management” on page 19-72 in the *Performance and Tuning Guide*.

max network packet size

Summary Information	
Name in previous release	**maximum network packet size**
Default value	512
Range of values	512–524288
Status	Static
Display level	Intermediate
Required role	System Administrator

The **max network packet size** parameter specifies the maximum network packet size that clients communicating with SQL Server can request.

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 **max network packet size** configuration parameter.

➤ ***Note***

While setting the value of **max network packet size** higher than **default network packet size** allows client processes to use larger packet sizes, these clients must explicitly request a larger packet size from within their applications.

The default value for **max network packet size** is 512 bytes. It must always be as large as, or larger than the **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 **max network packet size** and **additional network memory** 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 network memory** 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 3, since each connection needs three buffers.
- Add 2 percent for overhead
- Round the value to the next highest multiple of 2048.

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		* .02=1474
Add overhead	+ 1474	
Additional network memory	75202	
Round up to multiple of 2048	75776	

You should set **additional network memory** to 75,776 bytes.

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 **max network packet size** parameters 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 11-6 shows sample packet sizes.

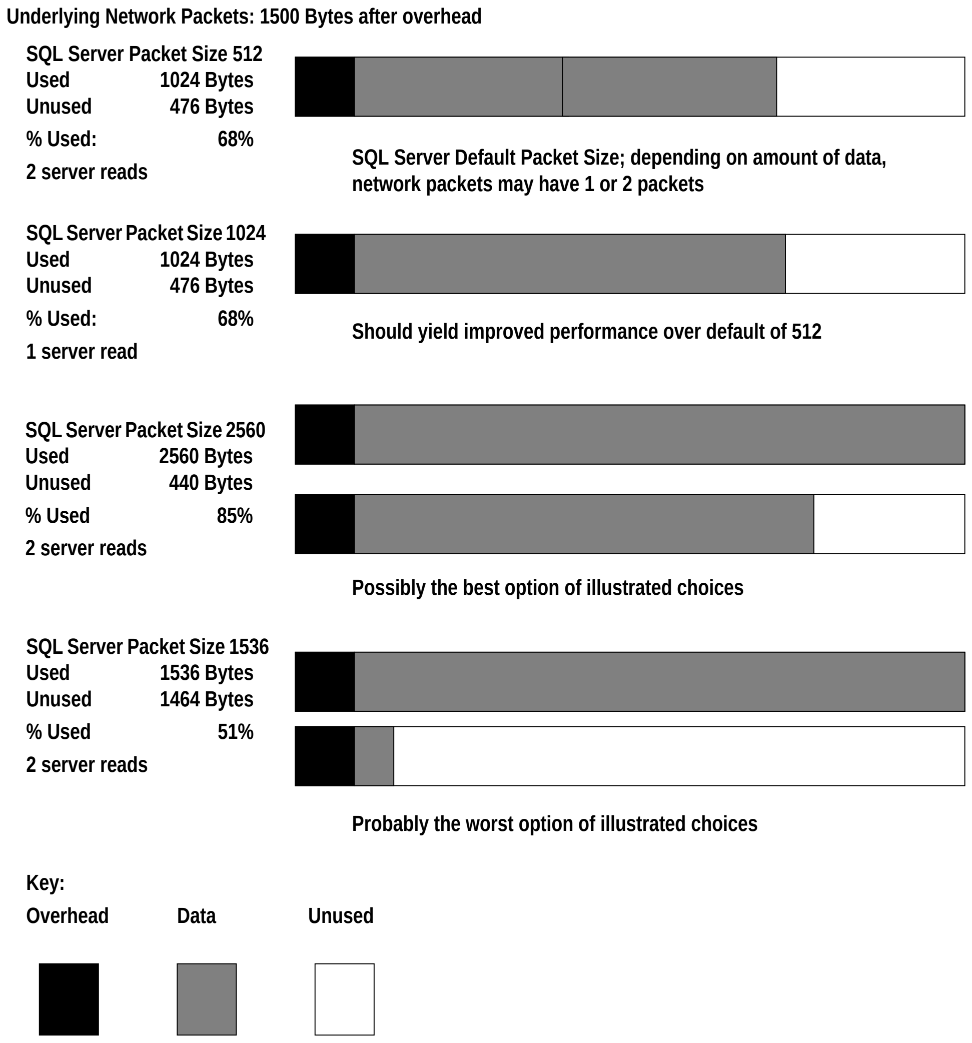

Underlying Network Packets: 1500 Bytes after overhead
SQL Server Packet Size 512
Used 1024 Bytes
Unused 476 Bytes
% Used: 68%
2 server reads
SQL Server Default Packet Size; depending on amount of data, network packets may have 1 or 2 packets
SQL Server Packet Size 1024
Used 1024 Bytes
Unused 476 Bytes
% Used: 68%
1 server read
Should yield improved performance over default of 512
SQL Server Packet Size 2560
Used 2560 Bytes
Unused 440 Bytes
% Used 85%
2 server reads
Possibly the best option of illustrated choices
SQL Server Packet Size 1536
Used 1536 Bytes
Unused 1464 Bytes
% Used 51%
2 server reads
Probably the worst option of illustrated choices
Key:
Overhead
Data
Unused

Figure 11-6: 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.

Use **sp_sysmon** to see how changing the **max network packet size** parameter affects network I/O management and task switching. For example, try increasing **max network packet size** and check **sp_sysmon** output to see how this affects **bcp** for large batches. See “Network Packet Received” on page 19-20, “Network Packet Sent” on page 19-21, and “Network I/O Management” on page 19-72 in the *Performance and Tuning Guide*.

max number network listeners

Summary Information	
Name in previous release	**cmaxnetworks**
Default value	15
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **max number network listeners** parameter specifies the maximum number of network listeners allowed by SQL Server at one time.

Each master port has one network listener. Generally, there is no need to have multiple master ports, unless your SQL Server needs to communicate over more than one network type. Some platforms support both socket and TLI (Transport Layer Interface) network interfaces. Refer to your Sybase installation and configuration guide for information on the network types supported by your platform. If your SQL Server does need to use multiple network types, configure **max number network listeners** to accommodate the number of network types you need.

number of remote connections

Summary Information	
Name in previous release	**remote connections**
Default value	20
Range of values	0–32767
Status	Static
Display level	Intermediate
Required role	System Administrator

The **number of remote connections** parameter specifies the number of logical connections that can be open to and from a single SQL Server at one time.

number of remote logins

Summary Information	
Name in previous release	**remote logins**
Default value	20
Range of values	0–32767
Status	Static
Display level	Intermediate
Required role	System Administrator

The **number of remote logins** parameter controls the number of active user connections from SQL Server to remote servers. You should set this parameter to the same (or a lower) value as **number of remote connections**.

number of remote sites

Summary Information	
Name in previous release	**remote sites**
Default value	10
Range of values	0–32767
Status	Static
Display level	Intermediate
Required role	System Administrator

The **number of remote sites** parameter determines the maximum number of remote sites that can simultaneously access SQL Server. Internally, **number of remote sites** determines the number of site handlers that can be active at any one time; all server accesses from a single site are managed with a single site handler.

remote server pre-read packets

Summary Information	
Name in previous release	**pre-read packets**
Default value	3
Range of values	0–32767
Status	Static
Display level	Intermediate
Required role	System Administrator

The **remote server pre-read packets** parameter determines the number of packets a site handler will "pre-read" during connections with remote servers.

All communication between two servers is managed through a single site handler, to reduce the required number of connections. The site handler can pre-read and keep track of data packets for each user process before the receiving process is ready to accept them.

The default value for **remote server pre-read packets** is 3, which is appropriate for almost all cases. Increasing the value uses more memory; decreasing the value can slow network traffic between servers.

tcp no delay

Summary Information	
Name in previous release	**T1610** (trace flag)
Default value	0 (off)
Valid values	0 (off), 1 (on)
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **tcp no delay** parameter toggles TCP (Transmission Control Protocol) packet batching. The default value is 0, which means that TCP packets are batched.

TCP normally batches small logical packets into single larger physical packets (by briefly delaying packets) in order to fill physical network frames with as much data as possible. This is intended to improve network throughput in terminal emulation environments where there are mostly keystrokes being sent across the network.

However, applications that use small TDS (Tabular Data Stream) packets may benefit from disabling TCP packet batching. To disable TCP packet batching, set **tcp no delay** to 1.

➤ ***Note***

Disabling TCP packet batching means that packets will be sent regardless of size; this will increase the volume of network traffic.

Operating System Resources

The parameters in this group relate to SQL Server’s use of operating system resources.

max async i/os per engine

Summary Information	
Name in previous release	**cnmaxaio_engine**
Default value	2147483647
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **max async i/os per engine** parameter specifies the maximum number of asynchronous disk I/O requests that can be outstanding for a single engine at one time.

Most operating systems limit the number of asynchronous disk I/Os that can be processed at any instant; some of these operating systems limit the number of asynchronous I/Os per operating system process, some limit the number of asynchronous I/Os per system, and some do both. If an application exceeds these limits, the operating system returns an error message. Because operating system calls are relatively expensive, it is inefficient for SQL Server to attempt to perform asynchronous I/Os that get rejected by the operating system.

To avoid this SQL Server maintains a count of the outstanding asynchronous I/Os per engine and per server; if an engine issues an asynchronous I/O that would exceed either **max async i/os per engine** or **max async i/os per server**, SQL Server delays the I/O until enough outstanding I/Os have completed to fall below whichever limit was exceeded.

In general, you should set **max async i/os per engine** to the limit allowed by your operating system for asynchronous disk I/O. However, if you are running other applications that use asynchronous I/O on the same machine, you may want to tune **max async i/os per engine** to a lower value so that SQL Server does not monopolize asynchronous I/O. (Refer to your operating system documentation for further information on asynchronous I/O. On some operating systems these limits can be tuned; on others they are hard coded.)

Because all I/Os (both asynchronous and synchronous) require a disk I/O structure, the total amount of disk I/O is limited by the value of **disk i/o structures**. Be sure to configure **max async i/os per engine** to a value lower than **disk i/o structures**. It is more efficient for SQL Server to delay an I/O than to have the I/O block because it cannot get a disk I/O structure.

If the limits for asynchronous I/O can be tuned on your operating system, you should make sure they are set high enough for SQL Server. There is no penalty for setting them as high as possible. At a minimum, they should be set to 50; for a heavily loaded SQL Server, 500 would be a more appropriate minimum.

max async i/os per server

Summary Information	
Name in previous release	**cnmaxaio_server**
Default value	2147483647
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **max async i/os per server** parameter specifies the maximum number of asynchronous disk I/O requests that can be outstanding for SQL Server at one time. This limit is not affected by the number of online engines per SQL Server; **max async i/os per server** limits the total number of asynchronous I/Os a server can issue at once, regardless of how many online engines it has.

Most operating systems limit the number of asynchronous disk I/Os that can be processed at any instant; some operating systems limit the number of asynchronous I/Os per operating system process, some limit the number of asynchronous I/Os per system, and some do both. If an application exceeds these limits the operating system returns an error message. Because operating system calls are relatively expensive, it is inefficient for SQL Server to attempt to perform asynchronous I/Os that get rejected by the operating system.

To avoid this, SQL Server maintains a count of the outstanding asynchronous I/Os per engine and per server; if an engine issues an asynchronous I/O that would exceed either **max async i/os per engine** or

max async i/os per server, SQL Server delays the I/O until enough that outstanding I/Os have completed to fall below the limit that was exceeded.

For example, assume an operating system limit of 200 asynchronous I/Os per system and 75 per process; assume a SQL Server with three online engines.

Engine	Number of I/Os Pending	Outcome
0	60	Engine 0 will delay any further asynchronous I/Os until total for server is under operating system **per-system** limit, then continue issuing asynchronous I/Os.
1	75	Engine 1 will delay any further asynchronous I/Os until per-engine total is under operating system **per-process** limit, then continue issuing asynchronous I/Os.
2	65	Engine 2 will delay any further asynchronous I/Os until total for server is under operating system **per-system** limit, then continue issuing asynchronous I/Os.

If either Engine 0 or Engine 2 completes an asynchronous I/O either one will then be able to issue more asynchronous I/Os, because by having completed a pending one the total for the whole server has gone below the operating system limit. However, even if Engines 0 and 2 complete pending asynchronous I/Os, Engine 1 will not be able to issue any further asynchronous I/Os until it drops below the operating system per-process limit of 75.

Use **sp_sysmon** to see if the **max async i/os per server** limit is delaying I/O on your system. If output for the sample period shows that SQL Server exceeded its limit for the number of outstanding disk I/O requests, you might consider raising **max async i/os per server.** See “Server Configuration Limit” on page 19-69 in the *Performance and Tuning Guide*.

In general you should set **max async i/os per server** to the limit allowed by your operating system for asynchronous disk I/O. However, if you are running other applications that use asynchronous I/O on the same machine, you may want to tune **max async i/os per server** to a lower value so that SQL Server does not monopolize asynchronous I/O. (Refer to your operating system documentation for further information on asynchronous I/O. On some operating systems, these limits can be tuned; on others, they are hard coded.)

Because all I/Os (both asynchronous and synchronous) require a disk I/O structure, the total amount of disk I/O is limited by the value of **disk i/o structures**. Be sure to configure **max async i/os per server** to a value lower than **disk i/o structures**: it is more efficient for SQL Server to delay an I/O than to have the I/O block because it cannot get a disk I/O structure.

If the limits for asynchronous I/O can be tuned on your operating system, make sure they are set high enough for SQL Server. There is no penalty for setting them as high as possible. At a minimum, they should be set to 50; for a heavily loaded SQL Server, 500 would be a more appropriate minimum.

o/s async i/o enabled

Summary Information	
Name in previous release	N/A
Default value	0
Valid values	0 (off), 1 (on)
Status	Read-only
Display level	Comprehensive
Required role	System Administrator

The **o/s async i/o enabled** parameter indicates whether or not asynchronous disk I/O has been enabled at the operating system level. This parameter is read-only, and cannot be configured.

In order for SQL Server to use asynchronous disk I/O, it must be enabled both at the operating system level and on SQL Server. See “allow sql server async i/o” on page 11-26 for information on enabling asynchronous disk I/O on SQL Server.

Refer to your operating system documentation for information on enabling asynchronous disk I/O at the operating system level.

o/s file descriptors

Summary Information	
Name in previous release	N/A
Default value	0
Range of values	Site-specific
Status	Read-only
Display level	Comprehensive
Required role	System Administrator

The **o/s file descriptors** parameter indicates the maximum per-process number of file descriptors configured for your operating system. This parameter is read-only and cannot be configured through SQL Server.

Many operating systems allow you to configure the amount of file descriptors available per process. See your operating system documentation for further information on this.

shared memory starting address

Summary Information	
Name in previous release	**mrstart**
Default value	0
Range of values	Platform-specific
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **shared memory starting address** parameter determines the virtual address at which SQL Server starts its shared memory region.

It is unlikely that you will ever have a need to reconfigure **shared memory starting address**. You should do so only after consulting with Sybase Technical Support.

Physical Memory

The parameters in this group configure your machine's physical memory resources.

additional network memory

Summary Information	
Name in previous release	**additional network memory**
Default value	0
Range of values	0–2147483647
Status	Static
Display level	Intermediate
Required role	System Administrator

The **additional network memory** parameter 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 value for **additional network memory** must be a multiple of 2048 bytes. The default value for **additional network memory** is 0, which means that no extra space has been allocated for large packets.

When SQL Server starts, a private pool of memory is allocated from SQL Server's memory region and is reserved for network packet allocations. Memory for all network packets, regardless of size, is allocated from this pool.

If a connection requests a packet size larger than **default network packet size,** SQL Server attempts to allocate the necessary buffers (read, write, and overflow). If successful, SQL Server swaps network buffers with the connection and deallocates the default buffers the connection began with. If buffers of the size requested cannot be allocated, SQL Server attempts to find a set of smaller buffers that are closest to the size requested.

SQL Server guarantees that every user connection will be able to log in at the default packet size. If you increase **max network packet size** but do not increase **additional network memory,** 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.

Increasing **additional network memory** may improve performance for applications that transfer large amounts of data. To determine the value for **additional network memory** 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 3, since each connection needs three buffers.
- Add 2 percent for overhead
- Round the value to the next highest multiple of 2048.

Memory allocated with **additional network memory** is added to the memory allocated by **total memory.** It does not affect other SQL Server memory uses.

lock shared memory

Summary Information	
Name in previous release	**T1611** (trace flag)
Default value	0 (off)
Valid values	0 (off), 1 (on)
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **lock shared memory** parameter disallows swapping of SQL Server pages to disk, and allows the operating system kernel to avoid the server’s internal page locking code. This can improve performance by reducing disk reads, which are expensive.

Not all platforms support shared memory locking. Even if your platform does support it, use of the **lock shared memory** parameter may fail because of incorrectly set permissions, insufficient physical memory, or other reasons.

total memory

Summary Information	
Name in previous release	**memory**
Default value	Platform-specific
Range of values	3850–2147483647
Status	Static
Display level	Intermediate
Required role	System Administrator

The **total memory** parameter sets the size of memory, in 2K units, that SQL Server allocates from the operating system. The default value of **total memory** varies from platform to platform. See the SQL Server installation and configuration guide for the value on your operating system.

Use the following worksheet to determine how much memory you will need for your configuration:

Category of Memory Use	Value for Your Server
executable code size +	
Static overhead (1MB) +	
Per-User Connection Amount **stack size** (platform specific) * **number of user connections** **stack guard size** @ 4,096 bytes * **number of user connections** (**default network packet size** * 3) * **number of user connections** +	 ________________ ________________ ________________
number of open databases * 644 bytes +	
number of locks * 32 bytes +	
number of devices * 45,056 bytes +	
size of procedure cache +	
total data cache size +	
number of extent i/o buffers * 16K	

= **total memory**

The more memory that is 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. See Chapter 8, “Configuring Memory,” for instructions about how to maximize the amount of **total memory** for SQL Server on your system.

Change the value of the **total memory** configuration parameter:

- When you change the amount of RAM on your machine
- When the pattern of use of your machine 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 boots, 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. (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.) 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.

Processors

The parameters in this group configure processors in an SMP environment.

max online engines

Summary Information	
Name in previous release	**max online engines**
Default value	1
Range of values	1–32
Status	Static
Display level	Intermediate
Required role	System Administrator

The **max engines online** parameter specifies the maximum number of SQL Server engines that can be online at any one time in an SMP environment. See Chapter 10, “Managing Multiprocessor Servers,” for a detailed discussion of how to set this parameter for your SMP environment.

At start-up, SQL Server boots with a single engine and completes all its initialization, including recovery of all databases. Its final task is to fork additional server engines. Each engine accesses common data structures in shared memory. As an engine completes or context-switches out of a task it scans the run queue for additional tasks to run. Normally, it performs system tasks once per clock tick.

When tuning the **max engines online** parameter:

- Never have more online engines than CPUs.
- Depending on overall system load (including applications other than SQL Server), you may achieve optimal throughput by leaving some CPUs free to run non-SQL Server processes.
- Better throughput can be achieved by running fewer engines with high CPU use rather than by running more engines with low CPU use.
- Scalability is application-dependent. You should do thorough benchmarking of your application to determine the best configuration of online engines.

min online engines

Summary Information	
Name in previous release	**min online engines**
Default value	1
Range of values	1–32
Status	Static
Display level	Intermediate
Required role	System Administrator

The **min online engines** parameter specifies the minimum number of SQL Server engines that should be online. This applies only to SMP environments.

SQL Server Administration

The parameters in this group relate to general SQL Server administration.

allow nested triggers

Summary Information	
Name in previous release	**nested trigger**
Default value	1 (on)
Valid values	0 (off), 1 (on)
Status	Static
Display level	Intermediate
Required role	System Administrator

The **allow nested triggers** parameter is a toggle that controls the use of nested triggers. When the value is set to 1, data modifications made by triggers can fire other triggers. A **set** option, **self_recursion**, controls whether triggers refire on data modification. Set **allow nested triggers** to 0 to disable nested triggers. The default is 1.

allow updates to system tables

Summary Information	
Name in previous release	**allow updates**
Default value	0 (off)
Valid values	0 (off), 1 (on)
Status	Dynamic
Display level	Comprehensive
Required role	System Security Officer

The **allow updates to system tables** parameter enables users with the System Security Officer role to make changes to the system tables and to create stored procedures than can modify system tables.

System tables include:

- All Sybase-supplied tables in the *master* database
- All tables in user databases that begin with "*sys*" and that have an ID value in the *sysobjects* table of less than or equal to 100

◆ ***WARNING!***

Incorrect alteration of a system table can result in database corruption and loss of data. Always use begin transaction when modifying a system table to protect against errors that could corrupt your databases. Immediately after finishing your modifications, disable allow updates to system tables; while it is enabled, any user with the System Administrator role can modify system tables or create stored procedures that can modify system tables.

Stored procedures and triggers you create while the **allow updates to system tables** parameter is set on are always able to update the system tables, even after the parameter has been set back to off. Whenever you turn the **allow updates to system tables** 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 **startserver** and **dataserver** in the SQL Server utility programs manual.

audit queue size

Summary Information	
Name in previous release	**audit queue size**
Default value	100
Range of values	1–65535
Status	Static
Display level	Intermediate
Required role	System Security Officer

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 parameter. There is a trade-off between performance and risk 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 affects overall system performance: user processes that generate audit records sleep if the audit queue is full.

Following 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 every 20 records at the most, and 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.

cpu accounting flush interval

Summary Information	
Name in previous release	**cpu flush**
Default value	200
Range of values	1–2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **cpu accounting flush interval** parameter specifies the amount of time, in machine clock ticks, that SQL Server waits before flushing CPU usage statistics for each user from *sysprocesses* to *syslogins*. (Note that this is measured in **machine** clock ticks, not SQL Server clock ticks.) This is used in chargeback accounting.

When a user logs onto SQL Server, the server begins accumulating figures for CPU usage for that user process in *sysprocesses*. Whenever a user logs off SQL Server, or the value of **cpu accounting flush interval** is exceeded, the accumulated CPU usage statistics are flushed from *sysprocesses* to *syslogins*. These statistics continue accumulating in *syslogins* until you clear the totals by using **sp_clearstats**. You display the current totals from *syslogins* by using **sp_reportstats**.

The value to which you set **cpu accounting flush interval** depends on the type of reporting you intend to do. If you intend to run reports on a monthly basis, setting **cpu accounting flush interval** to a relatively high value makes sense. This is because with infrequent reporting it is less critical that the data in *syslogins* be updated as often.

On the other hand, if you intend to do periodic ad hoc selects on the *totcpu* column in *syslogins*, to determine CPU usage by process, it makes sense to set **cpu accounting flush interval** to a lower value. Doing so increases the likelihood of the data in *syslogins* being up to date when you execute your selects.

Setting **cpu accounting flush interval** to low values may cause processes to be mistakenly identified as potential deadlock victims by the lock manager. When the lock manager detects a deadlock, it checks the amount of CPU time each of the competing processes has accumulated. The process with the lesser amount is chosen as the deadlock victim and is terminated by the lock manager. When **cpu accounting flush interval** is set to low values, the task handlers that store CPU usage information for processes are initialized more frequently, thus making processes appear as if they have accumulated less CPU time than they actually have. Because of this, the lock manager may select a process to be a deadlock victim when, in fact, that process has more accumulated CPU time than the competing process.

If you do not intend to report on CPU usage at all, you should set **cpu accounting flush interval** to its maximum value. This reduces the number of times *syslogins* is updated, and reduces the number of times its pages need to be written to disk.

cpu grace time

Summary Information	
Name in previous release	**ctimemax**
Default value	500
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **cpu grace time** parameter specifies the maximum amount of time (in milliseconds) that a user process can run without yielding the CPU before SQL Server preempts it and terminates it with a time-slice error.

When a process exceeds **cpu grace time** SQL Server "infects" it by removing the process from the internal queues. This prevents runaway processes from monopolizing the CPU. If any of your user processes become infected you may be able to temporarily fix the problem by increasing the value of **cpu grace time**. However, you must be sure that the problem really is a process that takes more than the current value of **cpu grace time** to complete, rather than a runaway process. This should be considered a workaround, not a permanent fix, and it may cause other complications. See "time slice" on page 11-97 for a more detailed discussion of process scheduling.

default database size

Summary Information	
Name in previous release	**database size**
Default value	2
Range of values	2–10000
Status	Static
Display level	Intermediate
Required role	System Administrator

The **default database size** parameter 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 parameter.

The default run value is 2MB. If most of the new databases on your SQL Server require more than 2MB, you may want to increase this value.

➤ ***Note***

You must increase the **database size** value if your *model* database becomes larger than 2MB, because the **create database** statement copies *model* to create a new user database.

default fill factor percent

Summary Information	
Name in previous release	**fillfactor**
Default value	0
Range of values	0–100
Status	Static
Display level	Intermediate
Required role	System Administrator

The **default fill factor percent** parameter determines how full SQL Server makes each index page when it is creating a new index on existing data, unless the user specifies some other value in the **create index** statement. The **default fill factor percent** affects performance, because SQL Server must take the time to split pages when they fill up. There is seldom a reason to change the **default fill factor percent** parameter, especially since you can override it in the **create index** command.

The **default fill factor percent** is used only when 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 fill factor percent** 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 **default fill factor percent** are 0 to 100.

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

A **default fill factor percent** of 0 creates clustered indexes with completely full data pages, and nonclustered indexes with completely full leaf pages. It is different from a value of 100 in that SQL Server leaves a comfortable amount of space within the index B-tree in both cases.

Other **default fill factor percent** values cause SQL Server to create new indexes with pages that are not completely full. For example, a **default fill factor percent** 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 **default fill factor percent** values cause each index to take more storage space.

event buffers per engine

Summary Information	
Name in previous release	N/A
Default value	100
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **event buffers per engine** parameter specifies the number of events per SQL Server engine that can be simultaneously monitored. Events are used in conjunction with SQL Server Monitor and a client tool for observing SQL Server performance. To monitor events you need to use SQL Server Monitor to enable event generation.

The value to which you should set **event buffers per engine** depends on the number of engines in your configuration, the level of activity on your SQL Server, and the kinds of applications you are running.

Setting **event buffers per engine** to low values may result in loss of event information. The default value, 100, is likely to be too low for most sites. Values of 2000 and above may be more reasonable for general monitoring. However, you need to experiment in order to determine the appropriate value for your site.

In general, setting **event buffers per engine** to higher values may reduce the amount of performance degradation that SQL Server Monitor causes for SQL Server.

If you do not intend to do any event monitoring, you should set **event buffers per engine** to 1.

Each event buffer uses 100 bytes of memory. To determine the total amount of memory used by a particular value for **event buffers per engine**, multiply the value by the number of SQL Server engines you have in your configuration. While 100 bytes is not a great deal of memory, the total amount used when setting **event buffers per engine** to high values can be substantial. For an eight-engine configuration, setting **event buffers per engine** to 20,000 would consume 1.56MB of memory.

housekeeper free write percent

Summary Information	
Name in previous release	N/A
Default value	1
Range of values	0–100
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **housekeeper free write percent** parameter specifies the maximum percentage by which the housekeeper task can increase database writes. Valid values range from 0 to 100.

For example, to stop the housekeeper task from working when the frequency of database writes reaches 25 percent above normal, set **housekeeper free write percent** to 25:

```
sp_configure "housekeeper free write percent", 25
```

When SQL Server has no user tasks to process, the housekeeper task automatically begins writing changed pages from cache to disk. These writes result in improved CPU utilization and a decreased need for buffer washing during transaction processing.

In applications that repeatedly update the same database page, the housekeeper may initiate some unnecessary database writes. Although these writes occur only during the server's idle cycles, they may be unacceptable on systems with overloaded disks.

By default, the **housekeeper free write percent** parameter is set to 1. This allows the housekeeper task to increase disk I/O by a maximum of 1 percent. This results in improved performance and recovery speed on most systems.

To disable the housekeeper task, set the value of **housekeeper free write percent** to 0:

```
sp_configure "housekeeper free write percent", 0
```

To allow the housekeeper task to work continuously, regardless of the percentage of additional database writes, set **housekeeper free write percent** to 100:

```
sp_configure "housekeeper free write percent", 100
```

Use **sp_sysmon** to see how this parameter affects the number of dirty buffers that reached the LRU before they could be written to disk and how many times I/O was already active on a buffer when it entered the wash area. See "Pool Turnover" on page 19-56 in the *Performance and Tuning Guide*.

It might also be helpful to look at the number of free checkpoints initiated by the housekeeper task. The *Performance and Tuning Guide* describes this output in "Number of Free Checkpoints" on page 19-65.

identity burning set factor

Summary Information	
Name in previous release	**identity burning set factor**
Default value	5000
Range of values	1 - 9999999
Status	Static
Display level	Intermediate
Required role	System Administrator

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 (or **shutdown with nowait**), 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 percent 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, and 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
```

identity grab size

Summary Information	
Name in previous release	N/A
Default value	1
Range of values	1–2147483647
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **identity grab size** parameter allows each SQL Server process to reserve a block of IDENTITY column values for inserts into tables that have an IDENTITY column.

This is useful if you are doing inserts and want all the inserted data to have contiguous IDENTITY numbers. For instance, if you are entering payroll data, and you want all records associated with a particular department to be located within the same block of rows, set **identity grab size** to the number of records for that department.

identity grab size applies to all users on SQL Server. Large **identity grab size** values result in large gaps in the IDENTITY column when many users insert data into tables that have IDENTITY columns.

In general, set **identity grab size** to a value large enough to accommodate the largest group of records you want to insert into contiguous rows.

i/o accounting flush interval

Summary Information	
Name in previous release	**i/o flush**
Default value	1000
Range of values	1–2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **i/o accounting flush interval** parameter specifies the amount of time, in machine clock ticks, that SQL Server waits before flushing I/O statistics for each user from *sysprocesses* to *syslogins*. (Note that this is measured in **machine** clock ticks, not SQL Server clock ticks.) This is used for chargeback accounting.

When a user logs onto SQL Server, the server begins accumulating I/O statistics for that user process in *sysprocesses*. Whenever the value of **i/o accounting statistics interval** is exceeded, or a user logs off SQL Server, the accumulated I/O statistics for that user are flushed from *sysprocesses* to *syslogins*. These statistics continue accumulating in *syslogins* until you clear the totals by using **sp_clearstats**. You display the current totals from *syslogins* by using **sp_reportstats**.

The value to which you set **i/o accounting flush interval** depends on the type of reporting you intend to do. If you intend to run reports on a monthly basis, setting **i/o accounting flush interval** to a relatively high value makes sense. This is because, with infrequent reporting, it is less critical that the data in *syslogins* be updated as often.

If you intend to do periodic ad hoc selects on the *totio* column *syslogins*, to determine I/O volume by process, it makes sense to set **i/o accounting flush interval** to a lower value. Doing so increases the likelihood of the data in *syslogins* being up to date when you execute your selects.

If you don't intend to report on I/O statistics at all, you should set **i/o accounting flush interval** to its maximum value. This reduces the number of times *syslogins* is updated, and reduces the number of times its pages need to be written to disk.

i/o polling process count

Summary Information	
Name in previous release	**cmaxscheds**
Default value	10
Range of values	1–2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **i/o polling process count** parameter specifies the maximum number of SQL Server processes run by SQL Server before the scheduler checks for disk and/or network I/O completions. Tuning **i/o polling process count** affects both the response time and throughput of SQL Server.

SQL Server checks for disk or network I/O completions:

- If the number of tasks run since the last time SQL Server checked for I/O completions equals the value for **i/o polling process count,** and
- At every SQL Server clock tick.

As a general rule, increasing the value of **i/o polling process count** may increase throughput for applications that generate lots of disk and network I/O. Conversely, decreasing the value may improve process response time in these applications, possibly at the risk of lowering throughput.

If your applications create both I/O-bound tasks and CPU-bound tasks, tuning **i/o polling process count** to low values (1–2) ensures that I/O-bound tasks get access to CPU cycles.

For OLTP applications (or any I/O-bound application with a lot of user connections and short transactions), tuning **i/o polling process count** to values in the range of 20–30 may increase throughput, though it may also increase response times.

When tuning **i/o polling process count**, you should consider three other parameters:

- **sql server clock tick length** (see “sql server clock tick length” on page 11-96), which specifies the duration of SQL Server’s clock tick in microseconds, and thus affects the amount of time the server will wait before checking for outstanding I/O

- **time slice** (see “time slice” on page 11-97), which specifies the number of milliseconds SQL Server’s scheduler allows a user process to run, is significant in determining how long CPU-bound tasks run before yielding the CPU
- **cpu grace time** (see “cpu grace time” on page 11-71), which specifies the maximum amount of time (in milliseconds) a user process can run without yielding the CPU before SQL Server preempts it and terminates it with a time-slice error

Use **sp_sysmon** to understand the effect of changing the **i/o polling process count** parameter. See “Average Network I/Os per Check” on page 19-13 and “Total Disk I/O Checks” on page 19-13 in the *Performance and Tuning Guide*.

lock promotion HWM

Summary Information	
Name in previous release	N/A
Default value	200
Range of values	2–2147483647
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **lock promotion HWM** (high water mark) parameter, together with the **lock promotion LWM** and **lock promotion PCT**, specifies the number of page locks permitted during a single scan of a table or index before SQL Server attempts to promote from page locks to a table lock. This value is effective server-wide.

For more information on lock promotion, see “Setting the Lock Promotion Thresholds” on page 11-9 in the *Performance and Tuning Guide*.

The value to which you set **lock promotion HWM** is server-wide; **it applies to all user tables on the server**. However, lock promotion occurs on a per-user, per-table basis. For instance, if you have **lock promotion HWM** set to 250, a particular user would have to acquire 250 page locks on a particular table before becoming eligible to acquire a table lock on that particular table.

The default value for **lock promotion HWM** (200) is likely to be appropriate for most applications. If you have a lot of small tables with clustered indexes, where there is contention for data pages, you may be able to increase concurrency for those tables by tuning **lock promotion HWM** to 80 percent of **number of locks**.

Because the effect of **lock promotion HWM** is server-wide, setting it to very low values is not a good idea in most cases. Doing so increases the chance of a particular user acquiring a table lock, thus denying other users access to that table for the duration of the table lock.

Lock promotion can also be configured at the per-object level by using the system procedure **sp_setpglockpromote**. If you have specific tables that require very idiosyncratic lock promotion tuning, it is best to do this with **sp_setpglockpromote**, rather than doing it at the server-wide level with **lock promotion HWM**. See **sp_setpglockpromote** in the *SQL Server Reference Manual*.

➤ ***Note***

When you configure **lock promotion HWM** make sure you do not set it to a value higher than the value of **number of locks**. If necessary, you can increase the value of **number of locks** to accommodate a high value for **lock promotion HWM**. Each lock requires 72 bytes of memory; if you significantly increase **number of locks,** you may also have to increase **total memory**.

Use **sp_sysmon** to see how changing the **lock promotion HWM** parameter affects the number of lock promotions. **sp_sysmon** reports on the number of exclusive page to exclusive table promotions and shared page to shared table promotions. See “Lock Promotions” on page 19-46 in the *Performance and Tuning Guide*.

lock promotion LWM

Summary Information	
Name in previous release	N/A
Default value	200
Range of values	2–value of **lock promotion HWM**
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **lock promotion LWM** parameter, together with the **lock promotion HWM** and the **lock promotion PCT**, sets the number of page locks permitted by a Transact-SQL command below which SQL Server never attempts to issue a table lock on the object.

See "Setting the Lock Promotion Thresholds" on page 11-9 of the *Performance and Tuning Guide* for more information on lock promotion.

The value to which you set **lock promotion LWM** is server-wide; **it applies to all user tables on the server**. Because the effect of **lock promotion LWM** is server-wide, setting it to very high values decreases the chance of a particular scan session acquiring a table lock (which uses more page locks for the duration of the transaction) and potentially exhausting all available locks on SQL Server. If this situation recurs, you may need to increase **number of locks.**

The default value for **lock promotion LWM** (200) is likely to be appropriate for most applications.

Lock promotion can also be configured at the per-object level by using the system procedure **sp_setpglockpromote**. If you have specific tables that require very idiosyncratic lock promotion tuning, it is best to do this with **sp_pglockpromote**, rather than doing it at the server-wide level with **lock promotion LWM**. See **sp_setpglockpromote** in the *SQL Server Reference Manual*.

lock promotion PCT

Summary Information	
Name in previous release	N/A
Default value	100
Range of values	1–100
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **lock promotion PCT** parameter, together with the **lock promotion HWM** and **lock promotion LWM**, sets the percentage of page locks permitted by a Transact-SQL command before SQL Server attempts to issue a table lock on the table, if the number of locks is between **lock promotion LWM** and **lock promotion HWM**.

See "Setting the Lock Promotion Thresholds" on page 11-9 of the *Performance and Tuning Guide* for more information on lock promotion.

The value to which you set **lock promotion PCT** is server-wide; **it applies to all user tables on the server**. Because the effect of **lock promotion PCT** is server-wide, setting it to very low values increases the chance of a particular user transaction acquiring a table lock. If this situation recurs, you may need to increase **number of locks.**

The default value for **lock promotion PCT** (100) is appropriate for most applications.

Lock promotion can also be configured at the per-object level by using the system procedure **sp_pglockpromote**. If you have specific tables that require very idiosyncratic lock promotion tuning, it is best to do this with **sp_pglockpromote**, rather than doing it at the server-wide level with **lock promotion PCT**. See **sp_setpglockpromote** in the *SQL Server Reference Manual*.

number of alarms

Summary Information	
Name in previous release	**cnalarm**
Default value	40
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **number of alarms** parameter specifies the number of alarm structures allocated by SQL Server.

The Transact-SQL command **waitfor** defines a specific time, time interval, or event for the execution of a statement block, stored procedure, or transaction. SQL Server uses alarms to execute **waitfor** commands correctly.

The number of alarms you need is determined by the needs of your applications, and the number of simultaneous instances of each application that uses **waitfor.** When SQL Server needs more alarms than are currently allocated, the message:

```
uasetalarm: no more alarms available
```

is written to the error log.

Each alarm structure uses 20 bytes of memory. If you significantly raise **number of alarms**, you should also adjust **total memory** accordingly.

number of extent i/o buffers

Summary Information	
Name in previous release	**extent i/o buffers**
Default value	0
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **number of extent i/o buffers** parameter specifies the number of extents (eight data pages) to be set aside as work buffers 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 index creation by speeding up disk reads and writes.

All the buffers are allocated to the first user who creates an index. If multiple users create indexes simultaneously, the first process uses extent I/Os; all others use normal page I/Os. Try to schedule creation of large indexes 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 **total memory** configuration parameter. A reasonable value for a server with 15MB of memory and a 20 percent procedure cache is 10, requiring 160K of memory (10 buffers * 8 pages per buffer * 2048 bytes per page).

Do not set the value for **number of extent i/o buffers** to more than 100.

number of mailboxes

Summary Information	
Name in previous release	**cnmbox**
Default value	30
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **number of mailboxes** parameter specifies the number of mailbox structures allocated by SQL Server. Mailboxes, used in conjunction with messages, are used internally by SQL Server for communication and synchronization between kernel service processes. Mailboxes are not used by user processes. Do not modify this parameter unless instructed to do so by Sybase Technical Support.

number of messages

Summary Information	
Name in previous release	**cnmsg**
Default value	64
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **number of messages** parameter specifies the number of message structures allocated by SQL Server. Messages, used in conjunction with mailboxes, are used internally by SQL Server for communication and synchronization between kernel service processes. Messages are not used by user processes. Do not modify this parameter unless instructed to do so by Sybase Technical Support.

number of open databases

Summary Information	
Name in previous release	**open databases**
Default value	12
Range of values	5–2147483647
Status	Static
Display level	Basic
Required role	System Administrator

The **number of open databases** parameter specifies the maximum number of databases that can be open simultaneously 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.

The default run value for **number of open databases** is 12. If SQL Server displays a message saying that you have exceeded the allowable number of open databases, increase this value. Each open database requires approximately 17K of memory. While 17K is not a huge amount of memory, it can add up to a non-trivial amount if you increase **number of open databases** by a lot. For instance, setting **number of open databases** to a value of 100 consumes 1.7MB of memory. The amount you increase **number of open databases** should be enough so that you no longer exceed the allowable number, but not so great that you run out of memory. The memory for **number of open databases** comes out of the same pool as your data cache. If you increase **number of open databases** without increasing **total memory** by a commensurate amount, you will have correspondingly less memory available for your data cache.

number of open objects

Summary Information	
Name in previous release	**open objects**
Default value	500
Range of values	100–2147483647
Status	Static
Display level	Basic
Required role	System Administrator

The **number of open objects** parameter 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 is insufficient, SQL Server displays a message.

Setting the **number of open objects** higher does not have a significant impact on performance or storage requirements. Therefore, do not hesitate to increase this value if SQL Server displays a message saying that you have exceeded the allowable number of open objects.

number of pre-allocated extents

Summary Information	
Name in previous release	**cpreallocext**
Default value	2
Range of values	0–31
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **number of pre-allocated extents** parameter specifies the number of extents (eight pages) allocated in a single trip to the page manager. Currently it is only used by **bcp** to improve performance when copying in large amounts of data. By default, **bcp** allocates one extent at a time, and writes an allocation record to the log each time. Setting

number of pre-allocated extents higher means that **bcp** allocates the specified number of extents each time it requires more space, and writes a single log record for the event.

Because an object may allocate more pages than actually needed you should keep the value for **number of pre-allocated extents** small if you are using **bcp** for small batches; if you are using **bcp** for large batches, you should increase the value of **number of pre-allocated extents**.

number of sort buffers

Summary Information	
Name in previous release	**csortbufsize**
Default value	0
Range of values	0–32767
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **number of sort buffers** parameter specifies the number of buffers used to hold pages read from input tables.

SQL Server does not allow processes to reserve more than half of its buffers; this determines the maximum value for **number of sort buffers**. If you are tuning **number of sort buffers** in conjunction with **number of extent i/o buffers** you should set **number of sort buffers** to eight times the value you set for **number of extent i/o buffers**.

You should do thorough benchmarking with a test server to determine the optimal value of **number of sort buffers** for your applications; note that you may encounter points of diminishing returns as you increase the value.

print deadlock information

Summary Information	
Name in previous release	**T1204** (trace flag)
Default value	0 (off)
Valid values	0 (off), 1 (on)
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **print deadlock information** parameter enables printing of deadlock information to the error log.

If you are experiencing recurring deadlocks, turning **print deadlock information** on provides you with information that can be useful in tracing the cause of the deadlocks.

➤ ***Note***

Turning **print deadlock information** on can seriously degrade SQL Server performance. For this reason, you should use it only when you are trying to determine the cause of deadlocks.

Use **sp_sysmon** output to determine if deadlocks are occurring in your running application. If they are, turn on the **print deadlock information** parameter to learn more about why they are occurring. See “Deadlocks by Lock Type” on page 19-44 in the *Performance and Tuning Guide*.

runnable process search count

Summary Information	
Name in previous release	**cschedspins**
Default value	2000
Range of values	0–2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **runnable process search count** parameter specifies the number of times an engine loops looking for a runnable task before relinquishing the CPU.

Multiple SQL Server engines always look for runnable tasks. At times there will not be any runnable tasks, and an engine could either relinquish the CPU or continue to look for a task to run. Setting **runnable process search count** higher causes the engine to hold the CPU for a longer time while the engine searches for a task to run. Setting the **runnable process search count** lower causes the engine to release the CPU sooner.

If your machine is a uniprocessor and your server is heavily loaded you may see some performance benefit from setting **runnable process search count** to 1. For CPU-intensive applications, you may be able to increase performance by increasing **runnable process search count** to 5.

For SQL Servers running on multiprocessor machines increasing **runnable process search count** to 500 may improve performance. For I/O intensive applications, values from 50 to 200 may be more appropriate. For CPU-intensive applications, values from 750–1250 may improve performance.

For both uniprocessor and multiprocessor machines, the above values may be best combined with setting the parameter **sql server clock tick length** to 50,000.

Use **sp_sysmon** to determine how the **runnable process search count** parameter affects SQL Server's use of CPU cycles, engine yields to the operating system, and blocking network checks. See "Engine Busy Utilization" on page 19-9, "CPU Yields by Engine" on page 19-11, and "Network Checks" on page 19-11 in the *Performance and Tuning Guide*.

partition groups

Summary Information	
Name in previous release	N/A
Default value	1024
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **partition groups** parameter specifies how many **partition group**s to allocate for SQL Server. Partition groups are internal structures that SQL Server uses to control access to individual partitions of a table.

A partition group is composed of 16 partition caches, each of which stores information about a single partition. All caches in a partition group are used to store information about the same partitioned table. If a table has fewer than 16 partitions, the unused partition caches are wasted. If a table has more than 16 partitions, it requires multiple partition groups.

The default value for the **partition groups** parameter is 64. This allows a maximum of 64 open partitioned tables and 1024 (64 times 16) open partitions. If you change the value of the **partition groups** parameter, you must **shutdown** and restart SQL Server for the new value to take effect.

SQL Server allocates partition groups to a table when you partition the table or when you access it for the first time after restarting SQL Server. If there are not enough partition groups for the table, you will not be able to access or partition the table.

partition spinlock ratio

Summary Information	
Name in previous release	N/A
Default value	10
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **partition spinlock ratio** parameter specifies the ratio of spinlocks to internal partition caches.

SQL Server manages access to table partitions using internal **partition group**s that each contain 16 partition caches. Each partition cache stores information about a partition (for example, the last page of the partition) that processes must use when accessing that partition. A partition spinlock is a simple, internal locking mechanism that prevents a process from accessing a partition cache currently used by another process. All processes must wait (or “spin”) until the lock is released.

The **partition spinlock ratio** parameter specifies the number of partition caches that each spinlock protects (partitions per spinlock), not the total number of spinlocks. The higher the value you specify for **partition spinlock ratio**, the lower the number of spinlocks protecting partition caches. By default, SQL Server systems are configured with **partition spinlock ratio** set to 32. This means that there is 1 spinlock for every 32 partition caches.

In theory, protecting one partition with one spinlock (by defining a **partition spinlock ratio** of 1) provides the least contention for a spinlock and the highest possible number of simultaneous partition operations. The suggested number of available spinlocks is 10 percent of the total number of partitions in use at any one time.

For example, if you configure the **partition groups** parameter to 80, the maximum number of open partitions is 1280. The suggested number of partition spinlocks in this case is 128, so you would set **partition spinlock ratio** to 10:

```
sp_configure "partition spinlock ratio", 10
```

You must shut down and restart SQL Server for any change to take effect.

➤ ***Note***

Decreasing the value of the **partition spinlock ratio** parameter may have little impact on the performance of SQL Server. The default setting for this parameter is correct for most servers.

Increasing the number of partition spinlocks beyond SQL Server's default value should reduce contention by only a very small amount (and may not result in a noticeable performance gain). Furthermore, each additional partition spinlock uses approximately 256 bytes of memory, which may affect other performance aspects of your server.

size of auto identity column

Summary Information	
Name in previous release	N / A
Default value	10
Range of values	1–38
Status	Dynamic
Display level	Intermediate
Required role	System Administrator

The **size of auto identity column** parameter sets the precision of IDENTITY columns that are automatically created with the **sp_dboption "auto identity"** option.

The maximum value that can be inserted into an IDENTITY column is $10^{\text{PRECISION}}$ -1. Once an IDENTITY column reaches its maximum value, all further **insert** statements return an error that aborts the current transaction.

If you reach the maximum value of an IDENTITY column, use the **create table** command to create a table identical to the old one, but with a larger precision for the IDENTITY column. Once you have created the new table, use the **insert** command or **bcp** to copy data from the old table to the new one.

sort page count

Summary Information	
Name in previous release	**csortpgcount**
Default value	0
Range of values	0–32767
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **sort page count** parameter specifies the maximum amount of memory a sort operation can use. (Sorts require about 50 bytes per row while processing the input table.) **sort page count** "borrows" pages from the default data cache for **each** sort operation. An optimal value for **sort page count** may be derived with the following formula;

(**number of sort buffers** x rows per data page) / 50

However, you will need to benchmark your applications to determine the best value for them. Be aware that you may encounter points of diminishing return as you increase **sort page count.**

➤ ***Note***

Database dumps may be incompatible on machines that have different values configured for the **sort page count** and **number of sort buffers** parameters. If a **create index** command is included in a transaction log dump, SQL Server uses the same values for these parameters that were used in the original sort.

To minimize this risk, **always dump database** after creating indexes on machines with large values configured for **sort page count** and **number of sort buffers**.

sql server clock tick length

Summary Information	
Name in previous release	**cclkrate**
Default value	Platform-specific
Range of values	Platform-specific min–1000000
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **sql server clock tick length** parameter specifies the duration of the server's clock tick, in microseconds. Both the default value and the minimum value are platform-specific. See your Sybase installation and configuration guide for the values for your platform.

In mixed-use applications with lots of CPU-bound tasks, decreasing the value of **sql server clock tick length** will help I/O-bound tasks. A value of 20,000 is reasonable for this. Shortening the clock tick length means that CPU-bound tasks will context switch more frequently per unit of time, and this allows other tasks greater access to the CPU. This may also marginally increase response times, because SQL Server runs its service tasks once per clock tick, and decreasing the clock tick length means that these service tasks will get run more frequently per unit of time.

Increasing **sql server clock tick length** favors CPU-bound tasks, because they can execute longer between context switches. A value of 1,000,000 is about the upper useful limit for this. If your applications are primarily CPU-bound, this may be a reasonable choice. However, any I/O-bound tasks will suffer as a result. This can be somewhat mitigated by tuning **time slice** (see "time slice" on page 11-97) and **cpu grace time** (see "cpu grace time" on page 11-71).

➤ ***Note***

Changing the value of **sql server clock tick length** can have dire effects on your SQL Server's performance. You should consult with Sybase Technical Support before resetting this value.

time slice

Summary Information	
Name in previous release	**time slice**
Default value	100
Range of values	50–1000
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **time slice** parameter sets the number of milliseconds that a user process is allowed to run by SQL Server's scheduler. If **time slice** is set too low, SQL Server may spend too much time switching processes. This is undesirable because context-switching is relatively expensive. If it is set too high, CPU-intensive processes may monopolize the CPU, and users may experience longer response times.

Figure 11-7 through Figure 11-9 illustrate three scenarios: a well-behaved process competing with other user processes for CPU cycles; a well-behaved process executing with no other processes competing for the CPU; and a poorly-behaved process terminating with a time-slice error. In this context, "well-behaved" refers to processes with hard-coded yield points that cause them to voluntarily relinquish the CPU at the end of the **time slice** interval. "Poorly behaved" refers to processes that fail to yield the CPU voluntarily.

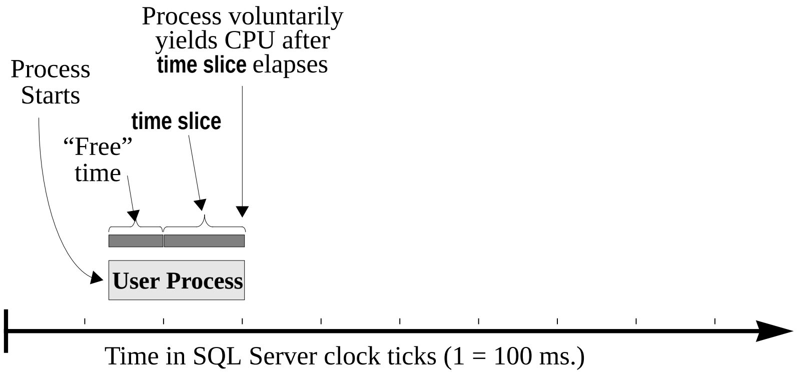

Process voluntarily yields CPU after time slice elapses
Process Starts
time slice
"Free" time
User Process
Time in SQL Server clock ticks (1 = 100 ms.)

Figure 11-7: Well-behaved process competing with other processes for CPU

Figure 11-7 shows a user process executing while other user processes are waiting for CPU time. The process starts between SQL Server clock ticks, and thus the time from when the process starts until the first SQL Server clock tick is "free," meaning that it doesn't count against the process. At the first SQL Server clock tick **time slice** begins, in this example set to the default value of 100 ms. Each time the process comes to a hard-coded yield point, it checks to see if **time slice** has been exceeded. If not, the process continues to execute. If so, the process voluntarily yields the CPU, and the next process in the run queue begins executing.

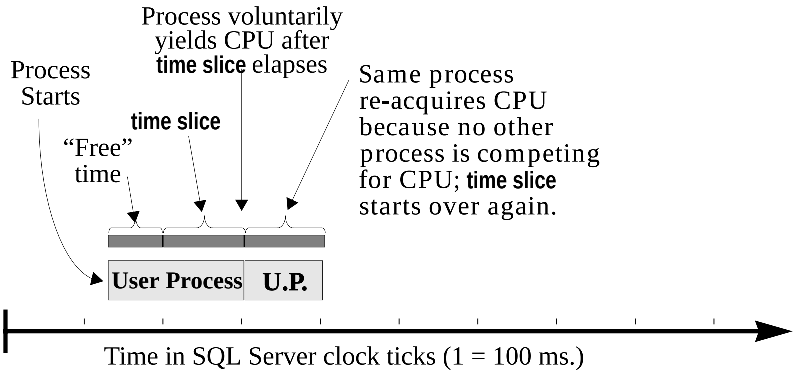

Process voluntarily yields CPU after time slice elapses
Process Starts
time slice
"Free" time
Same process re-acquires CPU because no other process is competing for CPU; time slice starts over again.
User Process
U.P.
Time in SQL Server clock ticks (1 = 100 ms.)

Figure 11-8: Well-behaved process with no other processes competing for CPU

Figure 11-8 shows the same user process executing when no other user processes are competing for CPU cycles. When the process comes to a hard-coded yield point and **time slice** has been exceeded, the process yields the CPU. However, in this example, there are no other processes competing for CPU cycles, so the process reacquires the CPU and continues executing. As soon as the process reacquires the CPU, the **time slice** interval starts again.

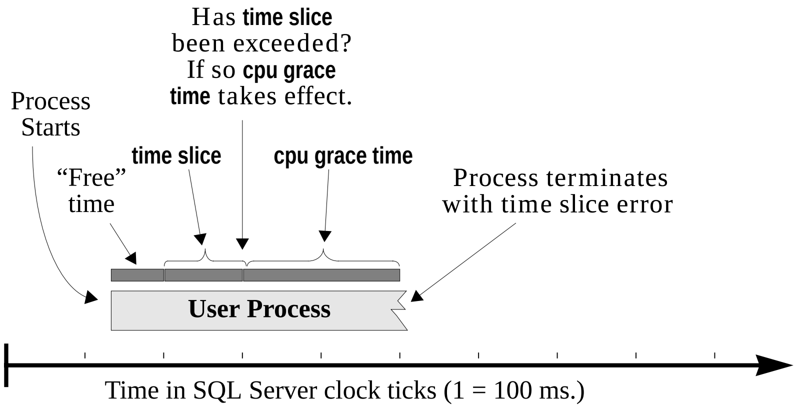


Figure 11-9: Poorly behaved process failing to yield CPU

Figure 11-9 shows a user process that fails to yield the CPU at the end of the **time slice** interval. As with the previous two examples, the process begins executing, and **time slice** starts at the first SQL Server clock tick. However, when **time slice** ends, the process fails to yield the CPU, and continues executing, regardless of whether or not other processes are waiting for the CPU. Once **time slice** has been exceeded, **cpu grace time** takes effect. (In this example, **cpu grace time** is set to 200 ms. See "cpu grace time" on page 11-71 for more details.) If the process is still executing at the end of **cpu grace time**, it terminates with a time slice error.

The default value for **time slice** is 100 ms., and there is seldom reason to change it. Setting **time slice** to a higher value increases the chances that CPU-bound tasks will monopolize the CPU. Setting it to a lower value may result in more processes terminating with time-slice errors.

If you tune **time slice**, you should take into consideration the values for the following parameters:

- **cpu grace time** specifies the maximum amount of time (in milliseconds) that a user process can run without yielding the CPU before SQL Server terminates it with a time slice error.
- **runnable process search count** specifies the number of times an engine loops looking for a runnable task before relinquishing the CPU.
- **sql server clock tick length** specifies the duration of the server's clock tick, in microseconds.
- **i/o polling process count** specifies the number of SQL Server processes the scheduler runs before checking for disk and/or network I/O completions.

Use **sp_sysmon** to determine how the **time slice** parameter affects voluntary yields by SQL Server engines. See "Voluntary Yields" on page 19-16 in the *Performance and Tuning Guide*.

upgrade version

Summary Information	
Name in previous release	**upgrade version**
Default value	1100
Range of values	0–2147483647
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

The **upgrade version** parameter reports the version of the upgrade utility that upgraded your master device. **Although this parameter is configurable, you should not reset it. Doing so may cause serious problems with SQL Server.** The reason it is configurable is that the upgrade utility needs to be able to modify the master device during an upgrade.

You can determine whether or not an upgrade has been done on your master device by using the **upgrade version** parameter without specifying a value:

```
sp_configure "upgrade version"
```

If the upgrade version does not match the value expected by SQL Server, only the System Administrator can log in.

User Environment

The parameters in this group configure user environments.

number of user connections

Summary Information	
Name in previous release	**user connections**
Default value	25
Range of values	5–2147483647
Status	Static
Display level	Basic
Required role	System Administrator

The **number of user connections** parameter 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 parameter but also on other system activity.

The maximum allowable number of connections (or file descriptors) per process is operating system dependent; see the Sybase installation and configuration guide.

The number of file descriptors available for SQL Server is stored in the global variable *@@max_connections*. You can report the maximum number of file descriptors your system can use with this command:

```
select @@max_connections
```

The return value represents the maximum number of file descriptors allowed by the system for your processes, 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 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 number of 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 52K of memory (this is platform-specific) as overhead per user connection. If you increase the **stack size** or **default network packet size** configuration parameters, the amount of memory per user connection increases also, reducing the amount of space available for the data and procedure caches.

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, fill out the following worksheet:

Category of values	Values for your server
Value of *@@max_connections*	
Minus	

Sum of values returned by following 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`
	________________
	________________
	________________

Maximum value to which **number of user connections** can be set =	

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. Following are processes requiring user connections:

- 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 is a good idea to estimate the maximum number of connections that will be used by SQL Server and to update **number of user connections** as you add physical devices or users to the system. Use **sp_who** periodically to determine the number of active user connections on your SQL Server.

User Connections on SMP Systems

In symmetric multiprocessor (SMP) systems, each database engine is a separate operating system process, with its own limit of maximum number of file descriptors. In order to determine the maximum number of user connections on an SMP system, first determine the maximum number of user connections per engine, and then multiply that by the number of engines you have online.

permission cache entries

Summary Information	
Name in previous release	**cfgcprot**
Default value	15
Range of values	1–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **permission cache entries** parameter determines the number of cache protectors per task.

Information on user permissions is held in the permission cache. When SQL Server needs to check permissions it looks first in the permission cache: If it does not find what it needs, it looks in the *sysprotects* table. In terms of response time, it is significantly faster if SQL Server finds the information it needs in the permission cache, and does not have to do a read on *sysprotects*.

However, SQL Server looks in the permission cache only when it is checking user permissions, but not when permissions are being granted or revoked. Whenever a permission is granted or revoked, the entire permission cache is flushed. This is because existing permissions have time stamps that become outdated when new permissions are granted or revoked.

If users on your SQL Server frequently do operations that require their grantable or revokable permissions to be checked, you may see a small performance gain by increasing the value of **permission cache entries**. This effect is not likely to be significant enough to warrant extensive tuning.

If users on your SQL Server frequently **grant** or **revoke** permissions, you should avoid setting **permission cache entries** to a large value. The space used for the permission cache would be wasted, as the cache gets flushed with each **grant** and **revoke**.

Each cache protector takes 28 bytes of memory.

stack guard size

Summary Information	
Name in previous release	**cguardsz**
Default value	4096
Range of values	0–2147483647
Status	Static
Display level	Comprehensive
Required role	System Administrator

The **stack guard size** parameter sets the size of the stack guard area. The stack guard area is an overflow stack of configurable size at the end of each stack. SQL Server allocates one stack for each user connection when it boots. These stacks are located contiguously in the same area of memory, with a guard area at the end of each stack. At the end of each stack guard area is a "guardword," which is a 4-byte structure with a known pattern.

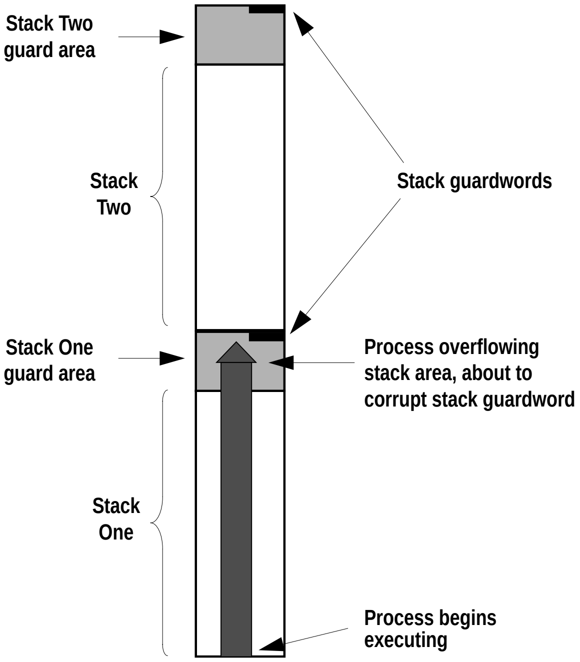


Figure 11-10:Process about to corrupt stack guardword

SQL Server periodically checks to see whether the stack pointer for a user connection has entered the stack guard area associated with that user connection's stack. If it has, SQL Server aborts the transaction, returns control to the application that generated the transaction, and generates Error 3626:

```
The transaction was aborted because it used too
much stack space.  Either use sp_configure to
increase the stack size, or break the query into
smaller pieces. spid: %d, suid: %d, hostname:
%.*s, application name: %.*s
```

SQL Server also periodically checks the guardword pattern to see if it has changed, thus indicating that a process has overflowed the stack boundary. When this occurs, SQL Server prints the messages to the error log:

```
kernel: *** Stack overflow detected: limit: 0x%lx sp: 0x%lx
kernel: *** Stack Guardword corrupted
kernel: *** Stack corrupted, server aborting
```

and shuts down. In the first message, "limit" is the address of the end of the stack guard area, and "sp" is the current value of the stack pointer.

In addition, SQL Server periodically checks the stack pointer to see whether it is completely outside both the stack and the stack guard area for the pointer's process. If it is, SQL Server shuts down, even if the guardword is not corrupted. When this happens, SQL Server prints the following messages to the error log:

```
kernel: *** Stack overflow detected: limit: 0x%lx sp: 0x%lx
kernel: *** Stack corrupted, server aborting
```

The default value for **stack guard size** should be appropriate for almost all applications. However, if you experience server shutdown from either stack guardword corruption or stack overflow, increase **stack guard size** by a 2K increment. Bear in mind that **each** configured user connection has a stack guard area; thus, when you increase **stack guard size**, you use up that amount of memory times the number of user connections you have configured. If you have a large number of user connections, this can add up to a significant amount of memory. Unless you also increase **total memory** by a corresponding amount, the memory you configure for **stack guard size** will come out of the same pool that is used for your data and procedure caches.

Long **where** clauses, long **select** lists, and deeply nested stored procedures can contribute to the probability of a stack overflow. However, rather than increasing **stack guard size** to remedy this, you should consider increasing **stack size** (see "stack size" on page 11-108). The stack guard area is intended as an overflow area, not as an extension to the regular stack.

SQL Server allocates stack space for each task by adding the values of the **stack size** and **stack guard size** parameters. This total has to be a multiple of SQL Server's page size, usually 2K or 4K. Refer to your operating system documentation for information on page size.

stack size

Summary Information	
Name in previous release	**stack size**
Default value	platform-specific
Range of values	Platform-specific min–2147483647
Status	Static
Display level	Basic
Required role	System Administrator

The **stack size** parameter specifies the size in bytes of the execution stacks used by each user process on SQL Server. See the Sybase installation and configuration guide for the minimum and default values for your platform. Stack size should always be specified in multiples of 2K. An execution stack is an area of SQL Server memory in which user processes keep track of their process context, as well as store local data.

Certain queries, particularly those with long **where** clauses, long select lists, and deeply nested stored procedures can all contribute to the probability of a stack overflow. Whenever a stack overflow occurs, SQL Server prints an error message and rolls back the transaction. See "stack guard size" on page 11-105 for more information on stack overflows. See *SQL Server Error Messages* for more information on specific error messages.

The two options for remedying this are to break these large queries into smaller queries, or to increase **stack size**. Changing **stack size** affects the amount of memory required for **each** configured user connection.

When considering changing **stack size**, you need to consider two issues:

- The value you have configured for **total memory**

  Because each user connection has its own execution stack, the amount of memory you specify for **stack size** gets multiplied by the number of user connections you've configured for SQL Server. This can add up to a significant amount of memory on SQL Servers with lots of user connections. If you increase **stack size** you will need to also increase **total memory** in order to maintain the same amount of memory available for your data cache. See "total memory" on page 11-64 for further information.

- The size and complexity of your largest queries

  If you have queries that exceed the size of the execution stack, it may make sense to rewrite them as a series of smaller queries. This is particularly true if there are only a small number of such queries, or if they are run infrequently.

There is no way to determine how much stack space a query will require without actually running the query. Stack space is preallocated at boot time. The space in a particular stack is used exclusively by a process until that process finishes executing, at which point the stack becomes available for another process.

Therefore, determining the appropriate value for **stack size** is an empirical process. You should test your largest and most complex queries using the default value for **stack size**. If they run without generating error messages the default is probably sufficient. If they do generate error messages, you should begin by increasing **stack size** by a small amount (2K). Rerun your queries and see if the amount you have added is sufficient. If it is not, continue until they run without generating error messages.

When you configure **stack size**, SQL Server checks the value to determine whether it is an even multiple of your SQL Server's page size, which is 2048 bytes on most platforms. If the value is not an even multiple, **sp_configure** rounds the value up to the nearest page-size multiple, and prints an informational message indicating the value.

systemwide password expiration

Summary Information	
Name in previous release	**password expiration interval**
Default value	0
Range of values	0–32767
Status	Dynamic
Display level	Intermediate
Required role	System Security Officer

The **systemwide password expiration** parameter sets the number of days that passwords remain in effect after they are changed. If **systemwide password expiration** 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 percent of the value of **systemwide password expiration** or 7 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. Users can change their passwords anytime 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 **systemwide password expiration** is 0 (no expiration). This parameter can be set only by System Security Officers.

user log cache size

Summary Information	
Name in previous release	N/A
Default value	2048
Range of values	2048–2147483647
Status	Static
Display level	Intermediate
Required role	System Administrator

The **user log cache size** parameter specifies the size (in bytes) for each user's log cache. There is one user log cache for each configured user connection. SQL Server uses these user log caches to buffer the user transaction log records, which reduces the contention at the end of the transaction log. The minimum (and default) value for **user log cache size** is 2048.

When a user log cache becomes full or another event occurs (such as when the transaction completes), SQL Server "flushes" all log records from the user log cache to the database transaction log. By first consolidating the log records in each user's log cache, rather than immediately adding each record to the database's transaction log, SQL Server reduces contention of processes writing to the log, especially for SMP systems configured with more than one engine.

You should not configure **user log cache size** larger than the maximum amount of log information written by an application's transaction. Since SQL Server flushes the user log cache when the transaction completes, any additional memory allocated to the user log cache is wasted. If no transaction in your server generates more than 4000 bytes of transaction log records, set **user log cache size** no higher than that value. For example:

```
sp_configure "user log cache size", 4000
```

If you configure **user log cache size** too high, it can increase the contention on the existing number of user log cache spinlocks (see the **user log cache spinlock ratio** parameter) and the probability of an internal event forcing the flush of the cache, wasting memory. If you set **user log cache size** too low, it can cause the user log cache to fill up and flush more than once per transaction, increasing the contention on the transaction log. (If the volume of transactions is low the amount of contention for the transaction log may not be significant.)

Use **sp_sysmon** to understand how this parameter affects cache behavior. See “ULC Flushes to Transaction Log” on page 19-28 and “Maximum ULC Size” on page 19-30 in the *Performance and Tuning Guide*.

user log cache spinlock ratio

Summary Information	
Name in previous release	N/A
Default value	20
Range of values	1–2147483647
Status	Static
Display level	Intermediate
Required role	System Administrator

The **user log cache spinlock ratio** parameter specifies the ratio of user log caches per user log cache spinlock. There is one user log cache for each configured user connection. A **spinlock** is a simple internal locking mechanism which prevents a second process from accessing the data structure currently used by another process. The second process must wait (or “spin”) until the lock is released. SQL Server uses spinlocks to protect the user log cache, since more than one process can access the contents of a user log cache.

The value you specify for this parameter defines the ratio of user log caches per spinlock. If you specify 5 for **user log cache spinlock ratio**, SQL Server allocates one spinlock for each 5 user log caches. For example:

```
sp_configure "user log cache spinlock ratio", 5
```

The actual number of spinlocks SQL Server allocates depends on the total number of user connections. The lower the value you specify for **user log cache spinlock ratio**, the higher the number of these spinlocks for your server.

However, you should be wary of setting this value too low, since each spinlock uses up to 256 bytes of memory. This additional memory requirement reduces the available memory for procedure and data caches, which may affect other performance aspects of your server. The default value for this parameter is 20; one spinlock for each 20 user connections configured for your server. If your server is configured with only one engine, SQL Server sets only one user log cache spinlock regardless of the number user connections.

Use **sp_sysmon** to understand how this parameter affects cache behavior. See “Spinlock Contention” on page 19-54 in the *Performance and Tuning Guide*.

12 Configuring Character Sets, Sort Orders, and Message Language

This chapter discusses SQL Server internationalization and localization support issues. It includes the following topics:

Introduction

Sybase provides both internationalization and localization support for international installations. **Internationalization** is the process of designing software products so that a single version can be adapted to different languages or regions, conforming to local requirements and customs without engineering changes. SQL Server is internationalized so that it can correctly process the characters used in different languages. The Sybase Character Sets product provides the character set definition files and sort order definition files required for data processing support for the major business languages in Western Europe, Eastern Europe, the Middle East, Latin America, and Asia. By default, SQL Server comes with the supported character sets and sort orders for the Western European languages.

Localization is the adaptation of an internationalized product to meet the requirements of one particular language or region, including providing translated system messages and correct formats for date, time, and currency. The Sybase Language Modules provide translated system messages and formats for the following languages: Chinese (Simplified), French, German, Japanese, and Spanish. By default, SQL Server comes with U.S. English message files.

This chapter briefly describes the character sets and language modules and summarizes the steps needed to change the default character set, sort order, or message language for SQL Server.

Character Sets and Sort Orders

If you require support for processing data in a language other than a Western European language, you need to purchase and install the Sybase Character Sets product.

Types of Internationalization Files

The files that support data processing in a particular language are called **internationalization files**. Several types of internationalization files come with SQL Server and the Sybase Character Sets product. These files are described below.

Table 12-1: Internationalization files

File	Location	Purpose and Contents
charset.loc	In each character set subdirectory of the *charsets* directory	Character set definition files that define the lexical properties of each character such as alphanumeric, punctuation, operand, upper or lower case. Used by the SQL Server to correctly process data.
**.srt*	In each character set subdirectory of the *charsets* directory	Defines the sort order for alphanumeric and special characters, including ligatures, diacritics, and other language-specific considerations.
**.xlt*	In each character set subdirectory of the *charsets* directory	Terminal-specific character translation 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 Chapter 13, “Converting Character Sets Between SQL Server and Clients,” and the SQL Server utility programs manual.

◆ ***WARNING!***

Do not alter any of the internationalization files. If you need to install a new terminal definition or sort order, contact your local Sybase distributor.

Character Sets Directory Structure

Figure 12-1 shows the directory structure for the Western European character sets that come with SQL Server. There is a separate subdirectory for each character set under the *charsets* directory. Within the subdirectory for each character set (for example, *cp850*) are the character set and sort order definition files and terminal-specific files.

If you load additional character sets from the Sybase Character Sets product, they will also appear under the *charsets* directory.

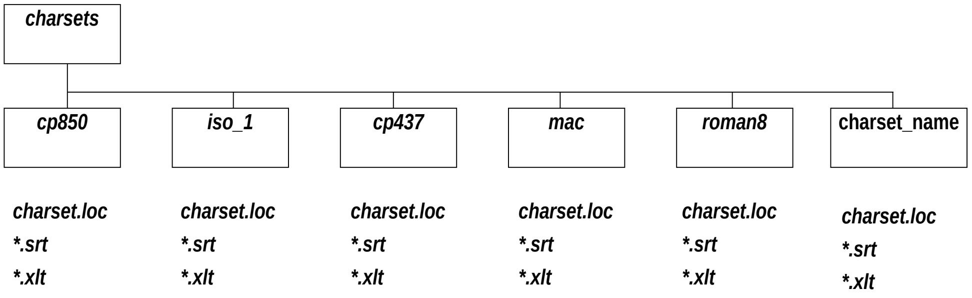


Figure 12-1: Character sets directory structure

Software Messages

By default, when you install SQL Server, U.S. English messages are loaded. International installations of SQL Server are supported with the Language Modules product that contain files with translated software messages and language or locale formats. These files, located in the locales subdirectory of the SQL Server installation directory, are called **localization files**.

Types of Localization Files

Several kinds of localization files are supplied with SQL Server and the Language Modules product, as shown in the following table.

Table 12-2: Localization files

File	Location	Purpose and Contents
locales.dat	In the *locales* directory	Used by client applications to identify the default message language and default character set.
server.loc	In the character set subdirectories under each language subdirectory in the *locales* directory	Software messages translated into the local language. Sybase products have product-specific **.loc* files. If an entry is not translated, that software 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 about the local date, time, and money formats.

◆ ***WARNING!***

Do not alter any of the localization files. If you need to alter any information in the files, contact your local Sybase distributor.

Software Messages Directory Structure

Figure 12-2 shows how localization files are arranged under the *locales* directory. Within the *locales* directory there is a subdirectory for each language installed. By default, there is always a *us_english* subdirectory. (On the PC platform, this directory is called english.) If you install language modules for additional languages, you will see subdirectories for those languages. Within each language, there are subdirectories for the supported character sets; for example, *cp850* is a supported character set for U.S. English. Software message files for each of the different Sybase products reside within the subdirectory for each character set. During the SQL Server installation, when you are prompted to select the languages you want to install on SQL Server, the install program lists the supported software message languages.

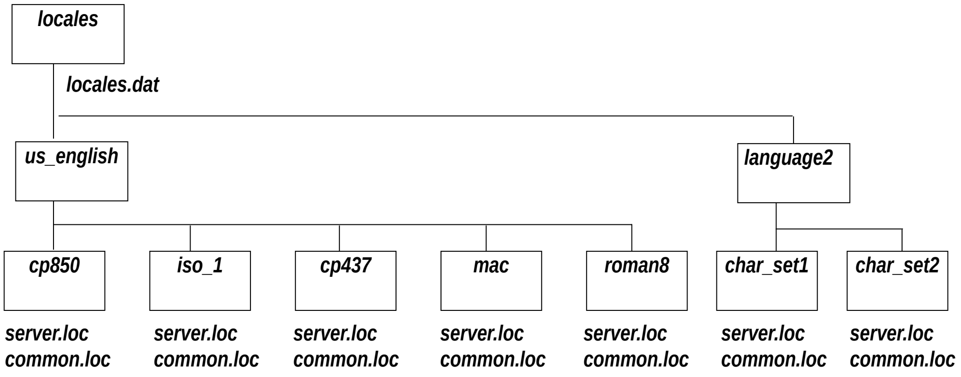


Figure 12-2: Messages directory structure

Changing the Default Character Set, Sort Order, or Message Language

A System Administrator can change the character set, sort order, or message language 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 more than one sort order may be available for a character set.

This section summarizes the steps to take before and after changing SQL Server's character set, sort order, and message language. For procedures on how to configure the message language, character set, and sort order, see the SQL Server installation and configuration guide.

Changing the Default Character Set

SQL Server can have only one **default character set**, the character set in which data is stored in its databases. When SQL Server is first installed, a default character set is specified.

◆ ***WARNING!***

Please read the following carefully and exercise extreme caution when changing the default character set of SQL Server.

Note the following when changing the default character set of SQL Server:

- You must convert any existing data to the new default character set.

  You must first copy the data out using **bcp**, and then change the default character set on SQL Server. Use **bcp**, with the appropriate flags for data conversion, to copy your data back into the server. See SQL Server utility programs manual for more information about **bcp**.

- Code conversions must be supported for the character sets.

  Code conversion between the character set of the existing data and the new default character set must be supported. If not, conversion errors will occur and the data will not be correctly converted. See Chapter 13, “Converting Character Sets Between SQL Server and Clients,” for more information about supported character set conversions.

- Some conversion errors may occur.

  Even if conversions are supported between the character sets, some conversion errors may occur because there are minor differences between the character sets, and some characters do not have equivalents in other character sets. Rows with problematic data may not get copied back into the database or data may contain partial or illegal characters.

If your existing data is 7-bit ASCII, it does not require conversion to the new default character set. You can change the default without first copying your data out of the server.

Changing the Default Sort Order

SQL Server can only have one **default sort order**, the collating sequence it uses to order data. When you consider changing the sort order for character data on a particular SQL Server, keep this in mind: All of your organization's SQL Servers should have the same sort order. A single sort order enforces consistency, and distributed processing is easier to administer.

Changing the sort order may result in invalid indexes and you may have to rebuild your indexes. For more information, see "If You Changed the Sort Order or Default Character Set" on page 12-9.

Getting Information About Sort Orders

The **sp_helpsort** system procedure displays SQL Server's default sort order, its character set, and a table of its primary sort orders. Its syntax is:

```
sp_helpsort
```

For more information about the different sort orders, see your Sybase installation and configuration guide.

Database Dumps and Configuration Changes

➤ ***Note***

Back up all databases on SQL Server before and after you change character sets or sort orders.

In most instances, you cannot reload your data from a database dump after reconfiguring the SQL Server default character set and sort order.

Do not use a database dump if the following is true:

- If your database contains 8-bit character data and you want your data to be converted to the new character set, do not load a database dump of your data into a SQL Server with the new default character set. SQL Server interprets all the data loaded as if it is in the new character set, and your data will be corrupted.
- If you are only making a change to the default sort order without changing the default character set, you cannot load a database from a dump that was performed before the sort order was

reconfigured. If you attempt to do so, an error message appears and the load is aborted.

- If the default character set is reconfigured, and either the old or the new sort order is not binary, you cannot load a database dump that was made before the character set was reconfigured.

Instead, use **bcp** to copy the data out of and into your databases.

You can use a database dump if the following is true:

- If your data does not have to be converted to the new character set and both the old and the new character sets use binary sort order, you can restore your database from backups that were made before the character set was reconfigured.

The Steps Involved

Several steps are involved in changing the SQL Server sort order, message language, or character set.

Preliminary Steps

Before you run the installation program to reconfigure SQL Server:

1. Dump all user databases and *master*. If you have made changes to *model* or *sybsystemprocs*, dump them also.
2. Load the language module if it is not already loaded (see the SQL Server installation and configuration guide for complete instructions).
3. If you are changing the SQL Server default character set, and your current databases contain non-7-bit data, copy the existing data out of your databases using **bcp**.

At this point, you can run the SQL Server installation program to configure languages, character sets, or sort orders, as well as other options.

Steps to Configure Languages, Character Sets, and Sort Orders

Use the SQL Server installation program to:

- Install or remove message languages and character sets included with SQL Server
- Change the default message language or character set

- Select a different sort order

See the SQL Server installation and configuration guide for instructions on using the installation program.

If you are adding a new character set that is not included with SQL Server, see the Sybase Character Sets documentation for complete instructions.

Final Steps

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 the SQL Server default character set, and your current databases contain non-7-bit data, copy your data back into your databases using **bcp** with the necessary flags to enable conversion.

If you changed SQL Server's default sort order or character set, also read "If You Changed the Sort Order or Default Character Set" on page 12-9.

Setting the User's Default Language

If an additional language is installed, users running client programs can run **sp_modifylogin** to set that language as their default language rather than SQL Server's default language.

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 12-10, and"Rebuilding Indexes with dbcc reindex" on page 12-11.

If you changed to a multibyte character set from any other character set (either multibyte or single-byte), you must upgrade any existing *text* values with **dbcc fix_text**. See "Upgrading text Data with dbcc fix_text" on page 12-12 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 18, "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* 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 so that it starts 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 [tab_name]
```

where *tab_name* is the optional name of a specific table. If *tab_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 and 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**. For 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 re-creates 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 also removed, 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 updates 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 19, “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 21, “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 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 Italian:

```
sp_addlanguage italian, italiano,
"gennaio,febbraio,marzo,aprile,maggio,giugno,luglio,agosto,settem
bre,ottobre,novembre,dicembre",
"genn,feb,mar,apr,mag,giu,lug,ago,sett,ott,nov,dic",
"lunedi,martedi,mercoledi,giovedi,venerdi,sabato,domenica",
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). Also, they must contain the right number of elements (12 for month strings, 7 for day-of-the-week strings.)

Valid values for the date formats are: *mdy*, *dmy*, *ymd*, *ydm*, *myd*, or *dym*. The *dmy* value indicates that the dates are in day / month / year order. This format affects only data entry; to change output format, you must use the **convert** function.

Server vs. Client Date Interpretation

Generally, date values are resolved on the client. When a user selects date values, SQL Server sends them to the client in internal format. The client uses the *common.loc* file and other localization files in the default language subdirectory of the *locales* directory on the client to convert the internal format into character data. For example, if the user's default language is Spanish, it looks for the *common.loc* file in the */locales/spanish/char_set* directory. It uses the information in the *common.loc* file it finds there to display, for example, *12 febrero 1995*.

Assume that the user's default language is set to *italian*, a language for which we currently do not provide a language module, and that date values in Italian have been added with **sp_addlanguage**. When the client connects to the server and looks for the *common.loc* file for Italian, it will not find the file. The client prints an error message and connects to the server. If the user then selects date values, the dates are displayed using the date strings for the server's default language. To display the date values added with **sp_addlanguage**, use the **convert** function to force the dates to be converted to character data at the server.

The following query generates a result set with the dates in the SQL Server default language:

```
select pubdate from titles
```

whereas the query below returns the date with the month names in Italian:

```
select convert(char(19),pubdate) from titles
```

13 Converting Character Sets Between SQL Server and Clients

This chapter describes how to convert between different character sets. It includes the following topics:

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 the utilities isql, bcp, and defncopy.

Conversion Paths Supported

SQL Server supports **character set conversion** among the character sets within the language groups shown in Table 13-1. Character set names in parentheses are the names used by SQL Server.

An additional character set, ASCII 7 (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 can pass between client and server unaltered. Other characters produce conversion errors.

Table 13-1: Supported character set conversions

Language Group	Character Sets
Cyrllic Script	Code Page 1251 (cp1251), Code Page 855 (cp855), Code Page 866 (cp866), ISO 8859-5 (iso88595), Koi 8 (koi8), Maccyr (mac_cyr)

Table 13-1: Supported character set conversions (continued)

Language Group	Character Sets
Eastern European	Code Page 1250 (cp1250), Code Page 852 (cp852), ISO 8859-2 (iso88592), Macee (mac_ee)
Greek	Code Page 1253 (cp1253), Code Page 869 (cp869), Greek 8 (greek8), ISO8859-7 (iso88597), Macgrk2 (macgrk2)
Japanese	DEC-Kanji (deckanji), EUC-JIS (eucjis), Shift-JIS (sjis)
Turkish	Code Page 1254 (cp1254), Code Page 857 (cp857), ISO 8859-9 (iso88599), Macturk (macturk), Turkish 8 (turkish8)
Western European	ASCII 8 (ascii_8), Code Page 437 (cp437), Code Page 850 (cp850), ISO 8859-1 (iso_1), Mac (mac), Roman 8 (roman8)
Arabic, Chinese (Simplified and Traditional), or Hebrew	No code set conversion is currently supported in these language groups.

Characters That Cannot Be Converted

In converting one character set to another, some characters may not 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 (code point 0xCE). This character does not exist in the ISO 8859-1 character set. If “Œ” exists in data that is 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 13-1 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.

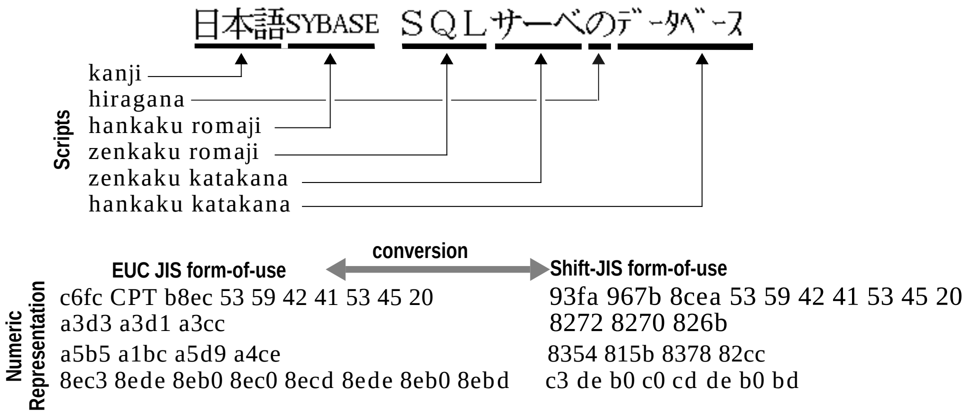


Figure 13-1: Comparison of EUC JIS and Shift-JIS encoding for Japanese characters

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 character set, 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.

When SQL Server encounters a conversion error in the data being entered, it 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, it replaces the bytes of the suspect characters with 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 13-5 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 Open Client DB-Library release 4.6 or later, 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 13-7 for details.

When a client requests a connection, SQL Server determines 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.
```

Next, SQL Server 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, and no message is sent.

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 *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 to "on" or "off"
- Start conversion to a specific character set
- Set error reporting to "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 off
  ```

  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 off 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 that 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 that was 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 13-2: Where character set conversion may be needed illustrates these different paths and the command line options that are 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 SQL 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 the **-J** option (**/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 that differs 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.

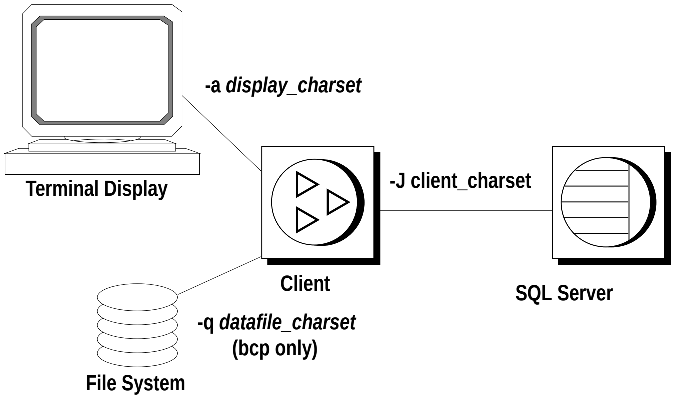


Figure 13-2: Where character set conversion may be needed

Managing Databases and Database Objects

14 Creating User Databases

This chapter explains how to create and modify user databases. It covers these topics:

Commands for Managing User Databases

Table 14-1 summarizes the commands that you will use to create, modify, and drop user databases and their transaction logs.

Table 14-1: Commands for managing user databases

Command	Task
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 from the default pool of database devices.
dbcc checktable(syslogs)	Reports the size of the log.
sp_logdevice	Specifies a device that will store the log when the current log device becomes full.
sp_helpdb	Reports information about a database's size and devices.

Table 14-1: Commands for managing user databases (continued)

Command	Task
sp_spaceused	Reports a summary of the amount of storage space used by a database.

Permissions for Managing User Databases

By default, only the System Administrator has **create database** permission. The System Administrator can grant permission to use the **create database** command. However, 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 and transfer ownership to another user, the System Administrator:

1. Issues the **create database** command
2. Switches to the new database with the **use** command
3. Executes the system procedure **sp_changedbowner**, as described in "Changing Database Ownership" on page 14-11.

The System Administrator can also grant permission to create databases. The user that receives **create database** permission must also be a valid user of the *master* database, since all databases are created while using *master.*

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 using the backups or dumps that are made regularly.

Permission to use the **alter database** or **drop database** command defaults to the Database Owner, and permission is automatically transferred with database ownership. **alter database** and **drop database** permission cannot be changed with **grant** or **revoke**.

Using the *create database* Command

Use the **create database** command to create user databases. You must have **create database** permission, and you must be a valid user of *master* (added with **sp_adduser**). Always type the **use master** command before creating a new database.

➤ ***Note***

Each time you enter the **create database** command, dump the *master* database. This makes recovery easier and safer in case *master* is later damaged. See Chapter 20, "Backing Up and Restoring the System Databases," for more information.

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 database devices listed in *master..sysdevices*:

```
   create database newpubs
```

You can control different characteristics of the new database by using the **create database** clauses:

- The **on** clause specifies the names of one or more database devices and the space allocation, in megabytes, for each database device. See "Assigning Space and Devices to Databases" on page 14-5 for more information.
- The **log on** clause places the **transaction log** (the *syslogs* table) on a separate database device with the specified or default size. See "Placing the Transaction Log on a Separate Device" on page 14-7 for more information.

- The **for load** option causes SQL Server to skip the page-clearing step during database creation. You can use this clause if you intend to load a dump into the new database as the next step. See "for load Option for Database Recovery" on page 14-10 for more information.
- The **with override** option allows SQL Servers on machines with limited space to maintain their logs on separate device fragments from their data. Use this option **only** when you are putting log and data on the same logical device. See "with override Option to create database" on page 14-11 for more information.

How *create database* Works

When a user with the required permission issues the **create database** statement, SQL Server:

- 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 all the remaining pages in the database device. If you are creating a database in order to load a database dump, the **for load** option skips page clearing (the page clearing step is performed after the load completes).

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 to "off" in *model*. If you want all of your new user databases to inherit particular options, change the options in *model* with the system procedure **sp_dboption**. See Chapter 15, "Setting Database

Options," for more information about changing database options. See Chapter 2, "System Databases," for more information about *model*.

Adding Users to Databases

After creating a new database, the System Administrator or Database Owner can manually add users to the database with **sp_adduser**. This task can be done with the help of the System Security Officer if new SQL Server logins are required. See the *Security Administration Guide* for details on managing SQL Server logins and database users.

Assigning Space and Devices to Databases

SQL Server allocates storage space to databases when users enter the **create database** or **alter database** commands. The **create database** command can specify one or more database devices, along with the amount of space on each that is to be allocated to the new database.

➤ ***Note***

You should also use the **log on** clause to place a production database's transaction log on a separate device. See "Placing the Transaction Log on a Separate Device" on page 14-7 for more information.

If you use the **default** keyword or if you omit the **on** clause altogether, SQL Server puts the database on one or more of the default database devices specified in *master..sysdevices*. See "Designating Default Devices" on page 6-7 for more information about the default pool of devices.

To specify a size (in this example, 4MB) for a database that is 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 a different amount of space on each. All the database devices named in **create database** must be listed in *sysdevices*. In other words, they must have been initialized with **disk init**. See

Chapter 6, “Initializing Database Devices,” for instructions about using **disk init**.

The following statement creates the *newdb* database and allocates 3MB on *mydata* and 2MB on *newdata*. As with the preceding example, the database and transaction log are not separated:

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

◆ ***WARNING!***

Unless you are creating a small or noncritical database, always place the log on a separate database device. Use the instructions under “Placing the Transaction Log on a Separate Device” on page 14-7 to create production databases.

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 each device. Then, SQL Server displays a message informing you how much space it has 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 do not specify a name), the **create database** command fails.

Default Database Size and Devices

If you omit the size parameter in the **on** clause, SQL Server creates the database with a default amount of space. This amount is the larger of the sizes specified by the **default database size** configuration parameter and the *model* database.

The size of *model* and the value of **default database size** are initially set to 2MB. To change the size of *model*, allocate more space to it with **alter database**. To change the **default database size** configuration parameter, use **sp_configure**. Changing **default database size** enables you to set the default size for new databases to any size between 2MB and 10,000MB. See “default database size” on page 11-72 for complete instructions.

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

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.

Estimating the Required Space

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

You can estimate the size of the tables and indexes for your database by using the **sp_estspace** system procedure or by calculating the value. See Chapter 5, "Estimating the Size of Tables and Indexes," in the *Performance and Tuning Guide* for instructions about estimating the size of objects.

Placing the Transaction Log on a Separate Device

The **log on** clause to **create database** places the **transaction log** (the *syslogs* table) on a separate database device. Placing the logs on a separate database device:

- Lets you use the **dump transaction** command rather than **dump database**, thus saving time and tapes.
- Lets you establish a fixed size for the log to keep it from competing for space with other database activity.
- 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. See Chapter 21, "Managing Free Space with Thresholds," for more information.
- Improves performance.
- Ensures full recovery from 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, noncritical databases, always place the log on a separate database device.

➤ ***Note***

If the log and its database share the same device, subsequent use of the system procedure **sp_logdevice** (as described on page 14-11) affects only future writes to the log. It 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.

To specify a size and device for the transaction log, use the **log on *device* = *size*** clause to **create database**. For example, the following statement creates the *newdb* database, allocates 8MB on *mydata* and 4MB on *newdata*, and places a 3MB transaction log on a third database device, *tranlog*:

```
create database newdb
on mydata = 8, newdata = 4
log on tranlog = 3
```

Estimating the Transaction Log Size

Two factors determine the size of the transaction log:

- The amount of update activity in the associated database
- The frequency of transaction log dumps

This is true whether you perform transaction log dumps manually or use threshold procedures to automate the task. As a rule of thumb, allocate to the log 10 percent to 25 percent of the space that you allocate to the database.

Inserts, deletes, and updates increase the size of the log. **dump transaction** decreases its size by writing committed transactions to disk and removing them from the log. Since **update** statements require logging both the “before” and “after” images of a row, applications that update many rows at once should plan on the transaction log being at least twice as large as the number of rows to be updated at the same time, or twice as large as your largest table. Or, you can **batch** the updates in smaller groups, performing transaction dumps between the batches.

In databases that have a lot of insert and update activity, logs can grow very quickly. To determine the required log size, periodically check the size of the log. This will also help you in choosing thresholds for the log and in scheduling the timing of transaction log dumps. To check the space used by a database's transaction log, first use the database. Then enter the command:

```
dbcc checktable(syslogs)
```

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

You can also use the following Transact-SQL statement to check on the growth of the log:

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

Repeat either command periodically to see how fast the log grows.

Default Log Size and Device

If you omit the *size* parameter in the **log on** clause, SQL Server allocates 2MB of storage on the specified log device. If you omit the **log on** clause entirely, SQL Server places the 2MB transaction log on the same database device as the data tables.

Moving the Transaction Log to Another Device

If you did not use the **log on** clause to **create database**, follow the instructions in this section 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.

The syntax for **sp_logdevice** is:

sp_logdevice *database_name, devname*

The database device 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 to another device, 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 a 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 18, "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 18, "Developing a Backup and Recovery Plan."

for load Option for Database Recovery

SQL Server generally clears all the unused pages in the database device when you create a new database. Clearing the pages can take several seconds or several minutes to complete, depending on the size of the database and the speed of your system.

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** that skips the page clearing step. This creates 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 in the dumped database. See Chapter 19, "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 Server on machines that have limited space to maintain their logs on separate device fragments from their data. Use this option **only** when you put log and data on the same logical device. This is not recommended practice, but it may be the only option available on machines with limited storage, especially if you need to get databases back online following a hard disk crash.

You will still be able to dump your transaction log, but if you experience a media failure, you will not be able to access the current log, since it is 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.

In the following example, the 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.

Changing Database Ownership

A System Administrator might want to create the user databases and give ownership of them to another user after completing some of the initial work. The system procedure **sp_changedbowner** changes the ownership of a database. The procedure must be executed by the System Administrator in the database where the ownership will be changed. The syntax is:

```
sp_changedbowner loginame [, true ]
```

The following example makes the user "albert" the owner of the current database and drops the 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. See the *Security Administration Guide* for more information about changing 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.

alter database Summary

When your database or transaction log grows to fill all the space you allocated with **create database**, you can use the **alter database** command to add storage. You can add space for database objects or for the transaction log or for both. You can also use **alter database** 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. For more information, see "Changing Database Ownership" on page 14-11. **alter database** permission cannot be changed with **grant** or **revoke**.

alter database Syntax

To extend a database, and to specify where storage space is to be added, use the full **alter database** syntax:

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

In its simplest form, **alter database** adds 1MB from the default database devices. If your database separates log and data, the space you add is used only for data. Use **sp_helpdevice** to find names of database devices that are in your default list.

To add 1MB from a default database device to the *newpubs* database, type:

```
alter database newpubs
```

The **on** and **log on** clauses in the **alter database** command operate 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 use **alter database** to extend the *master* database, you can extend it only on the master device. The minimum increase you can specify is 1MB (512 2K pages).

To add 3MB to the space allocated for the *newpubs* database on the database device named *pubsdata1*, type:

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

If SQL Server cannot allocate the requested size, it allocates as much as it can on each database device with a minimum allocation of .5MB (256 2K pages) per device. When **alter database** completes, it prints messages telling you how much space it allocated; for example:

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

Check all messages to make sure the requested amount of space was added.

The following command adds 2MB to the space allocated for *newpubs* on *pubsdata1*, 3MB on a new device, *pubsdata2*, and 1MB 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

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

The *for load* Clause

Use the **for load** clause only after **create database for load** to re-create the space allocation of the database being loaded into the new database from a dump. See Chapter 19, “Backing Up and Restoring User Databases,” for a discussion of duplicating space allocation when loading a dump into a new database.

drop database Summary

Use the **drop database** command to remove a database from SQL Server, thus deleting the database and all the objects in it. This command:

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

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

drop database Syntax

The syntax of this command is:

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

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

```
drop database newpubs, newdb
```

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.

System Tables That Manage Space Allocation

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—1/2MB, or 256 contiguous 2K pages. The first page of each allocation unit is the allocation page. It does not contain database rows like the other pages, but 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 it.

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 bitmask 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 the devices named in the **log on** statement and 3 for the data segment that holds the system segment ($2^0 = 1$) + default segment ($2^1 = 2$).

Some possible values for segments containing data or logs are listed in the following table:

Table 14-2: Segment values

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

Values higher than 7 indicate user-defined segments. The *segmap* column is explained more fully in the segments tutorial section in Chapter 16, “Creating and Using Segments.”

The *lstart*, *vstart*, and *size* Columns

- *lstart* column – contains the starting page number in 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.
- *size* column – contains the number of contiguous 2K pages that are assigned to the same database. The ending logical address of this portion of the database can be determined by adding the values in *lstart* and *size*.
- *vstart* column – contains the address where the piece assigned to this database begins. The upper 4 bits store the virtual device number (*vdevno*), and the lower 4 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.

Getting Information About Database Storage

This section explains how to determine which database devices are currently allocated to databases and how much space the database uses.

Database Device Names and Options

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. The status column lists the database options. These options are described in Chapter 15, "Setting Database Options."

If you are using the named database, **sp_helpdb** also reports on the segments in the database and the devices named by the segments. See Chapter 16, "Creating and Using Segments," for more information.

When you use **sp_helpdb** without arguments, it reports information about all 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
```

Space Used

To get a summary of 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
```

Table 14-3 describes the columns in the report.

Table 14-3: Columns in sp_spaceused output

Column	Description
database_name	Gives the name of the database being examined.
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 created in the database. (Space is allocated to database objects inside a database in increments of 1 extent, or 8 pages, at a time.)
data, index_size	Shows how much space has 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.

The sum of the values in the *unused*, *index_size*, and *data* columns should be the figure in the *reserved* column. Subtract *reserved* from *database_size* to get the amount of unreserved space. This space is available for new or existing objects that grow beyond the space that has been reserved for them.

By running **sp_spaceused** regularly, you can monitor the amount of database space available. For example, if the *reserved* value is close to the *database_size* value, you are running out of space for new objects. If the *unused* value is also small, you are 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 different from the results of running **select count(*)** on the table. This is because **sp_spaceused** computes the value with the built-in function **rowcnt**. That function uses values that are stored in the allocation pages. These values are not updated regularly, however, so they can be very different for tables with a lot of activity. The **update statistics**, **dbcc checktable**, and **dbcc checkdb** commands 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 particularly a problem if the transaction log is not on a separate device—in which case it competes 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.

15 Setting Database Options

This chapter describes how to use database options. It covers the following topics:

What Are Database Options?

Database options control many different aspects of database behavior, such as:

- The behavior of transactions
- Defaults for table columns
- Restrictions to user access
- Performance of recovery and **bcp** operations
- Log behavior

System Administrators and Database Owners can use database options to configure the settings for an entire database. This differs from **sp_configure** parameters, which affect the entire server, and **set** options, which affect only the current session or stored procedure.

The *sp_dboption* Procedure

Use the system procedure **sp_dboption** to change settings for an entire database. The options remain in effect until they are changed. The **sp_dboption** procedure:

- Displays a complete list of the 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), you must execute **sp_dboption** 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.

Database Option Descriptions

All users with access to the *master* database can execute the **sp_dboption** procedure with no parameters 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
identity in nonunique index
no chkpt on recovery
no free space acctg
read only
select into/bulkcopy
single user
trunc log on chkpt
trunc. log on chkpt.
```

For a report on which options have been set in a particular database, execute the system procedure **sp_helpdb** in that database.

The following sections describe each database option in detail.

abort tran on log full

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.

allow nulls by default

Setting **allow nulls by default** to **true** changes the default null type of a column from **not null** to **null**, in compliance with the SQL 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.

auto identity

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 columns with the **select *** statement. To retrieve it, you must explicitly mention the column name, *SYB_IDENTITY_COL*, in the select list.

To set the precision of the automatic IDENTITY column, use the **size of auto identity** configuration parameter.

dbo use only

While the **dbo use only** option is set to “on” (**true**), only the Database Owner can use the database.

ddl in tran

Setting the **ddl in tran** option to **true** allows the following commands to be used inside a user-defined transaction:

Table 15-1: DDL commands allowed in transactions

alter table (clauses other than **partition** and **unpartition** are allowed)	**create default**	**drop default**	**grant**
	create index	**drop index**	**revoke**
	create procedure	**drop procedure**	
	create rule	**drop rule**	
	create schema	**drop table**	
	create table	**drop trigger**	
	create trigger	**drop view**	
	create view		

Data definition statements must lock system tables for the duration of a transaction, which can result in performance problems. Use them only in short transactions.

The following commands cannot be used in a user-defined transaction under any circumstances:

Table 15-2: DDL commands not allowed in transactions

alter database
alter table...partition
alter table...unpartition
create database
disk init
dump database
dump transaction
drop database
load transaction
load database
select into
truncate table
update statistics

identity in nonunique index

The **identity in nonunique index** option automatically includes an IDENTITY column in a table's index keys so that all indexes created on the table are unique. This database option makes logically non-unique indexes internally unique, and allows those indexes to be used to process updatable cursors and isolation level 0 reads.

The table must already have an IDENTITY column for the **identity in nonunique index** option to work, either from a **create table** statement or by setting the **auto identity** database option to **true** before creating the table.

Use **identity in nonunique index** if you plan to use cursors and isolation level 0 reads on tables that have non-unique indexes. A unique index ensures that the cursor will be positioned at the correct row the next time a **fetch** is performed on that cursor.

no chkpt on recovery

The **no chkpt on recovery** option is set to "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 to "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 is not rerun unnecessarily, changes the sequence number on the database. If the sequence number on the secondary database has been changed, a subsequent dump of the transaction log from the primary database cannot 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.

no free space acctg

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.

read only

The **read only** option means that users can retrieve data from the database, but cannot modify anything.

select into/bulkcopy

The **select into/bulkcopy** option must be set to **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 that do not have indexes.

SQL Server performs minimal logging for these commands, recording only page allocations and deallocations, but not the actual changes that are made on the data pages.

You do not have to set the **select into/bulkcopy** option to **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 will not 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 are not recoverable from transaction logs.

Just setting the **select into/bulkcopy** option does not block log dumping, but making minimally logged changes to data does block the use of a regular **dump transaction.** However, you can still use **dump transaction...with no_log** and **dump transaction...with truncate_only.**

By default, the **select into/bulkcopy** option is turned off in newly created databases. To change the default situation, turn this option on in the *model* database.

single user

When **single user** is set to **true**, only one user at a time can access the database.

trunc log on chkpt

The **trunc log on chkpt** option means that the transaction log is truncated (committed transactions are removed) when the **checkpoint** checking process occurs (usually more than once per minute) if 50 or more rows have been written to the log. The log is **not** truncated if less than 50 rows were written to the log, or if the Database Owner runs the **checkpoint** command manually.

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 you 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.

Changing 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.

To 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 an ambiguous value for *optname*, 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 at a time can be turned on. You cannot change database options inside a user-defined transaction.

Viewing the Options on a Database

Use the **sp_helpdb** procedure to determine which options are set for a particular database. **sp_helpdb** lists each of the active database options in the "status" column of its output.

The following example shows that the **read only** option is turned on in *mydb*:

```
sp_helpdb mydb
```

```
name   db_size  owner  dbid  created       status
-----  -------  -----  ----  ------------  ----------------------
mydb   2.0 MB    sa     5    Mar 05, 1995  read only

device_fragments      size    usage           free kbytes
-----------------     ------  ------------  -------------
master                2.0 MB  data and log            576

device                         segment
------------------------------ ------------------------------
master                         default
master                         logsegment
master                         system

name    attribute_class attribute  int_value char_value
         comments
------- --------------- ---------- --------- --------------------------
         --------
pubs2   buffer manager  cache name      NULL cache for database mydb
         NULL
```

To display a summary of the database options for all databases, use **sp_helpdb** without specifying a database:

```
              sp_helpdb
```

```
name                db_size         owner         dbid
   created        status
-------------------  --------------  -------------  -----
   --------------  ---------------------------

mydb                2.0 MB           sa            5
   May 10, 1995   read only

master              3.0 MB           sa            1
   Jan 01, 1995   no options set

model               2.0 MB           sa            3
   Jan 01, 1995   no options set

sybsystemprocs     2.0 MB           sa           4
   Mar 31, 1995   trunc log on chkpt

tempdb              2.0 MB           sa            2
   May 04, 1995   select into/bulkcopy
```

16 Creating and Using Segments

This chapter introduces the system procedures and commands for using segments, or named collections of devices, in databases. It includes the following topics:

See also Chapter 13, "Controlling Physical Data Placement," in the *Performance and Tuning Guide* for information about how segments can improve system performance.

Commands and Procedures for Using Segments

Table 16-1 summarizes the SQL Server commands and procedures for using segments.

Table 16-1: Commands and procedures for managing segments

Command	Used To
sp_addsegment	Define a segment in a database
create table and **create index**	Create a database object on a segment
sp_dropsegment	Remove a segment from a database or remove a single device from the scope of a segment
sp_extendsegment	Add additional devices to an existing segment

Table 16-1: Commands and procedures for managing segments (continued)

Command	Used To
sp_placeobject	Assign future space allocations for a table or an index to a specific segment
sp_helpsegment	Display the segment allocation for a database or data on a particular segment
sp_helpdb	Display the segments on each database device. See Chapter 14, “Creating User Databases,” for examples.
sp_help	Display information about a table, including the segment on which the table resides
sp_helpindex	Display information about a table’s indexes, including the segments on which the indexes reside

What Is a Segment?

Segments are named subsets of the database devices that are 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. Using segments can increase SQL Server performance and give the System Administrator or Database Owner increased control over the placement, size and space usage of database objects.

You create segments within a particular database to describe the database devices that are already allocated to the database. Each SQL Server database can contain up to 32 segments, including the system-defined segments (see “System-Defined Segments” on page 16-3). Before assigning segment names, you must initialize the database devices with **disk init** and then make them available to the database with **create database** or **alter database**. See Chapter 14, “Creating User Databases” for information about **create database** and **alter database**.

System-Defined Segments

When you first create a database, SQL Server automatically creates three segments in the database, as described in Table 16-2.

Table 16-2: System-defined segments

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

If you create a database on a single database device, the *system*, *default*, and *logsegment* segments label the same device. If you use the **log on** clause to place the transaction log on a separate device, the segments resemble those shown in Figure 16-1.

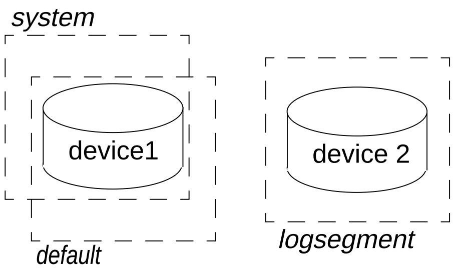


Figure 16-1: System-defined segments

Why Use Segments?

When you add a new device to a database with **alter database,** SQL Server places the new device in a default pool of space (the database's *default* and *system* segments). This increases the total space available to the database, but it does not determine which objects will occupy that new space. Any table or index might grow to fill the entire pool of space, leaving critical tables with no room for expansion. It is also possible for several heavily used tables and indexes to be placed on a single physical device in the default pool of space, resulting in poor I/O performance.

Segments provide a way for System Administrators and Database Owners to control the placement of database objects on database devices. When you create an object on a segment, the object can use all the database devices that are available in the segment, but no other devices. You can use this feature to control the space that is available to individual objects. More importantly, you can use it to improve SQL Server performance.

The following sections describe how to use segments to control disk space usage and to improve performance. "Moving a Table to Another Device" on page 16-7 explains how to move a table from one device to another using segments and clustered indexes.

Controlling Space Usage

If you assign noncritical objects to a segment, those objects cannot grow beyond the space available in the segment's devices. Conversely, if you assign a critical table to a segment, and the segment's devices are not available to other segments, no other objects will compete with that table for space.

When the devices in a segment become full, you can extend the segment to include additional devices or device fragments as needed. Segments also allow you to use thresholds to warn you when space becomes low on a particular database segment.

Segments and Thresholds

Thresholds monitor the amount of free space in a database segment, and can perform actions automatically when the space becomes full. Each database that stores its transaction log on a separate device from its data has at least one threshold: the last-chance threshold. If you create additional segments for data, you can create new threshold procedures for each segment. See Chapter 21, "Managing Free Space with Thresholds," for more information on thresholds.

Improving Performance

In a large, multidatabase and/or multidrive SQL Server environment, careful attention to the allocation of space to databases and the placement of database objects on physical devices can enhance system performance. Ideally, each database has exclusive use of database devices, that is, it does not share a physical disk with another database. In most cases, you can improve performance by placing heavily used database objects on dedicated physical disks, or by “splitting” large tables over several physical disks.

The following sections describe these ways to improve performance. See also the *Performance and Tuning Guide* for more information about how segments can improve performance.

Separating Tables, Indexes, and Logs

Generally, placing a table on one physical device, its nonclustered indexes on a second physical device, and the transaction log on a third physical device can speed up performance. Using separate physical devices (disk controllers) reduces the time required to read or write to the disk, since it usually reduces disk head travel. If you cannot devote entire devices in this way, at least restrict all nonclustered indexes to a dedicated physical device.

The **log on** extension to **create database** (or **sp_logdevice**) handles the placement of the transaction log on a separate physical disk. Use segments to place tables and indexes on specific physical devices. See “Assigning Database Objects to Segments” on page 16-10 for information about placing tables and indexes on segments.

Splitting Tables

Splitting a large, heavily used table across devices on separate disk controllers can improve the overall read performance of a table. When a large table exists on multiple devices, it is more likely that small, simultaneous reads will take place on different disks. Figure 16-2 shows a table that is split across the two devices in its segment.

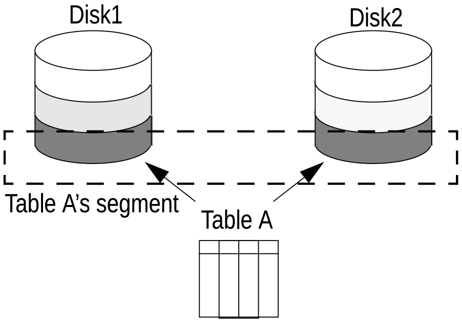


Figure 16-2: Partitioning a table over physical devices

You can split a table across devices using three different methods, each of which requires the use of segments:

- Use table partitioning if the table does not have a clustered index
- Use the partial load method if the table has a clustered index
- Separate the text chain from other data if the table contains *text* or *image* datatypes

Partitioning Tables

Partitioning a table creates multiple page chains for the table, and distributes those page chains over all the devices in the table's segment (see Figure 16-2). Partitioning a table increases insert performance as well as read performance, since multiple page chains are available for insertions.

Before you can partition a table, you must create the table on a segment that contains the desired number of devices. The remainder of this chapter describes how to create and modify segments. See "Partitioning and Unpartitioning Tables" in the *Performance and Tuning Guide* for information about partitioning tables using the **alter table** command.

You cannot partition tables that have clustered indexes.

Partial Loading

If you want to split a table that has a clustered index, you can use **sp_placeobject** with multiple **load** commands to load different parts of the table onto different segments. This method can be difficult to execute and maintain, but it provides a way to split tables and their clustered indexes over physical devices. See "Placing Existing Objects on Segments" on page 16-12 for more information and syntax.

Separating text and image Columns

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's other data. 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 and base table data on a separate physical devices can improve performance. See "Placing Text Pages on a Separate Device" on page 16-15 for more information and syntax.

Moving a Table to Another Device

You can also use segments to move a table from one device to another using the **create clustered index** command. 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. Therefore, you can completely move a table by dropping its clustered index (if one exists), and creating or re-creating a clustered index on the desired segment. See "Creating Clustered Indexes on Segments" on page 16-15 for more information and syntax.

Creating Segments

Two preliminary steps are needed to create a segment in 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**. This automatically adds the new device to the database's *default* and *system* segments, as described under "System-Defined Segments" on page 16-3.

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

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

where:

segname is any valid identifier. It is used in **create table** and **create index** statements to create those objects on the segment and, therefore, on the device named in *devname*. Give segments names that identify what they are used for, and use extensions like "_seg."

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

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

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

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

Changing the Scope of Segments

To perform the tasks described in “Why Use Segments?” on page 16-3, you will also need to manage the scope of segments—the number of database devices to which a segment points. You can:

- Extend the scope of a segment, by making it point to an additional device or devices, or
- Reduce the scope of a segment, by making it point to fewer devices.

Extending the Scope of Segments

You may need to extend a segment if the database object or objects assigned to the segment run out of space. The stored procedure **sp_extendsegment** extends the size of a segment by 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
```

The word “default” is a keyword, so if you want to extend the *default* segment in your database, you must place the word “default” in quotes:

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

Automatically Extending the Scope of a Segment

If you use **alter database** to add space on a database device that is new to the database, the *system* and *default* segments are automatically extended to include this the new space. Thus, the scope of the *system* and *default* segments is automatically extended each time you add a new device to the database.

If you use **alter database** to assign additional space on an existing database device, all the segments mapped to the existing device are automatically extended to include the new device fragment. For example, assume that you have initialized a 4MB device named *newdev* and have allocated 2MB of the device to *mydata* and assigned it to the *testseg* segment:

```
alter database mydata on newdev = 2
sp_addsegment testseg, mydata, newdev
```

If you alter *mydata* later to use the remaining space on *newdev*, the remaining space fragment is automatically mapped to the *testseg* segment:

```
alter database mydata on newdev = 2
```

See “A Segment Tutorial” on page 16-19 for more examples about how SQL Server automatically assigns new device fragments to segments.

Reducing the Scope of a Segment

You may need to reduce the scope of a segment if it includes database devices that you want to reserve exclusively for other segments. For example, if you add a new database device that is to be used exclusively for one table, you will want to reduce the scope of the *default* and *system* segments so that they no longer point to the new device.

Use the **sp_dropsegment** procedure to drop a single database device from a segment, reducing the segment’s scope. The following syntax removes a single database device from a segment:

sp_dropsegment *segname*, *dbname*, *device*

With three arguments, **sp_dropsegment** does not drop the segment, but drops only the given *device* from the scope of devices spanned by the segment. You can also use **sp_dropsegment** to remove an entire segment from the database, as described under “Dropping Segments” on page 16-16.

The following example removes the database device *pubs_dev2* from the scope of *bigseg*:

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

Assigning Database Objects to Segments

This section explains how to assign new or existing database objects to user-defined segments in order to:

- Restrict new objects to one or more database devices
- Place a table and its index on separate devices to improve performance
- Split an existing object over multiple database devices

This section uses the techniques described in “Creating Segments” and “Changing the Scope of Segments” on page 16-8.

Creating New Objects on Segments

To place a new object on a segment, first follow the instructions for “Creating Segments” on page 16-7 to create the new segment. You may also want to change the scope of this segment (or other segments) so that it points only to the desired database devices. Remember that when you add a new database device to a database, it is automatically added to the scope of the *default* and *system* segments.

After you have defined the segment in the current database, use the **create table** or **create index** command with the optional **on *segment_name*** clause to create the object on the segment. The syntax is:

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

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

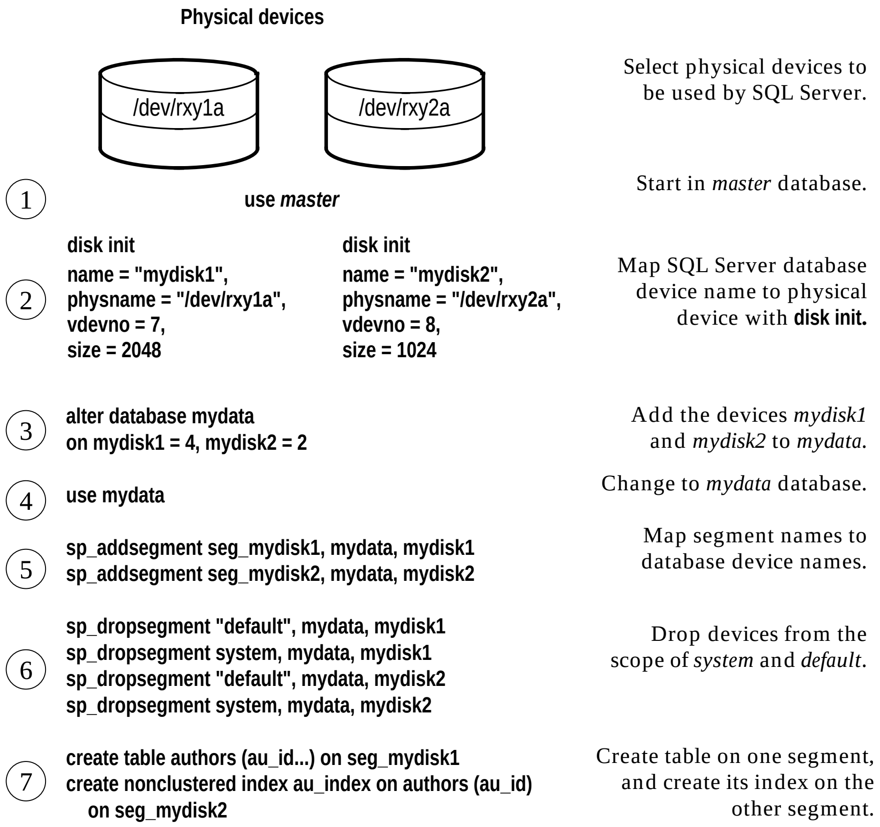
➤ ***Note***

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 “Creating Clustered Indexes on Segments” on page 16-15.

Example: Creating a table and index on separate segments

Figure 16-3: Creating objects on specific devices using segments summarizes the sequence of Transact-SQL commands used to create tables and indexes on specific physical disks:

1. Start by using the master database.
2. Initialize the physical disks.
3. Allocate the new database devices to a database.
4. Use the database.
5. Create two new segments that each point to one of the new devices.
6. Reduce the scope of the *default* and *system* segments so that they do not point to the new devices.
7. Create the objects, giving the new segment names.



Select physical devices to be used by SQL Server.

```
1   use master
```

Start in *master* database.

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

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

```
3   alter database mydata
    on mydisk1 = 4, mydisk2 = 2
```

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

```
4   use mydata
```

Change to *mydata* database.

```
5   sp_addsegment seg_mydisk1, mydata, mydisk1
    sp_addsegment seg_mydisk2, mydata, mydisk2
```

Map segment names to database device names.

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

Drop devices from the scope of *system* and *default*.

```
7   create table authors (au_id...) 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 16-3: Creating objects on specific devices using segments

Placing Existing Objects on Segments

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 new segment it specifies. The syntax is:

`sp_placeobject` ***`segname, objname`***

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. See “Creating Clustered Indexes on Segments” on page 16-15 for instructions on moving a table.

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.

Example: Splitting a table and its clustered index across physical devices

Performance can be improved for high-volume, multi-user applications when large tables are split across segments that are located on separate disk controllers.

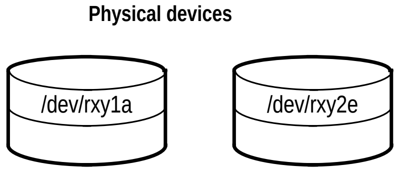
Figure 16-4: Splitting a large table across two segments summarizes the process of splitting a table across two segments, as shown in the following steps:

1. Begin by using the *master* database.
2. Initialize the devices with **disk init**.
3. Assign both devices to the *mydata* database with **alter database**.
4. Use the *mydata* database by typing the **use** command.
5. Create 3 segments. The first two should each point to one of the new devices. Extend the scope of the third segment so that it labels both devices.
6. Drop the *system* and *default* segments from both devices.
7. Create the table and its clustered index on the first segment.
8. Load half the table's data onto the first segment.
9. Use **sp_placeobject** to cause all further allocations of disk space to occur on the second segment.
10. Load the remaining data onto the second segment.
11. Once the data has been loaded onto the second segment, use **sp_placeobject** again to place the table on the segment that spans both devices.

➤ ***Note***

The order of steps 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, creating a clustered index would cause the entire object to migrate to the second segment. If the index is created after the table is placed on *seg_bothdisks*, the allocation of disk space is unpredictable.

In the preceding example, the balance of disk allocation may change over time if the table is updated frequently. To guarantee that the speed advantages are maintained, it may be necessary to drop and re-create the table at some point.



Select physical devices to be used by SQL Server.

1

```
use master
```

Start in *master* database.

2

```
disk init
name = "mydisk1",
physname = "/dev/rxy1a",
vdevno = 7,
size = 2048
```

```
disk init
name = "mydisk2",
physname = "/dev/rxy2e",
vdevno = 8,
size = 2048
```

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

3

```
alter database mydata
on mydisk1 = 4, mydisk2 = 4
```

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

4

```
use mydata
```

Change to *mydata* database.

5

```
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.

6

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

Drop devices from the scope of *system* and *default*.

7

```
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.

8 [use **bcp** to load half of the rows]

Load half of the rows.

9

```
sp_placeobject segmydisk2, authors
```

Place the object on the second segment.

10 [use **bcp** to load the rest of the rows]

Load the rest of the rows.

11

```
sp_placeobject seg_bothdisks, authors
```

Place the table on the segment that spans both disks.

Figure 16-4: Splitting a large table across two segments

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**, pages that were previously written on the old device remain allocated, but all new allocations take place on the new segment.

Creating Clustered Indexes on Segments

The bottom or leaf level of a clustered index contains the data. Therefore, by definition, a table and its clustered index are on the same segment. If you create a table on one segment and 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 was created. This provides a quick and easy mechanism for moving a table to other devices in your database.

The syntax for creating a clustered index on a segment is:

```
create [unique] clustered index index_name
   on [[database.]owner.]table_name (column_name
        [, column_name]...)
   [with {fillfactor = x, ignore_dup_key, sorted_data,

     [ignore_dup_row | allow_dup_row]}]
   on segment_name
```

See "Segments and Clustered Indexes" on page 16-24 for an example of this command.

Dropping Segments

When you use the stored procedure **sp_dropsegment** with only a segment name and the database name, the named segment is dropped from the database. However, you cannot drop a segment as long as database objects are still assigned to it. You must assign the objects to another segment (as described under “Assigning Database Objects to Segments” on page 16-10) or drop the objects first and then drop the segment. Use the commands described under “Getting Information About Segments” on page 16-16 to determine which objects are still assigned to a segment.

The syntax for dropping a segment is:

```
sp_dropsegment segname, dbname
```

You can also use **sp_dropsegment** to remove a database device from the scope of a segment. See “Reducing the Scope of a Segment” on page 16-9 for details.

➤ ***Note***

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

If you drop all 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.” Always use **sp_addsegment “default”**, ***dbname, devname*** to map a device to the default segment unless you plan to assign it to another segment.

Getting Information About Segments

Four 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_help** and **sp_helpindex** display information about tables and indexes, including the segment to which the object is assigned.

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*:

```
sp_helpsegment seg1

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

device                 size           free_pages
---------------------- -------------- -----------
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.

For example:

```
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
```

sp_help and *sp_helpindex*

When you execute **sp_help** and **sp_helpindex** in a database, and give a table's name, the system procedure displays information about which segment(s) stores the table or its indexes.

For example:

```
sp_helpindex authors
```

```
index_name     index_description                       index_keys
------------   -----------------------------------     ----------
au_index       nonclustered located on seg_mydisk2     au_id
```

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 under “A Segment Tutorial” on page 16-19.

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

➤ ***Note***

See “System Tables That Manage Space Allocation” on page 14-15 for more information about the *segmap* column and the system tables that manage storage.

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 that has already been assigned a fragment of that device, all 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 automatically 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 will 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* segments are both mapped to *bigdevice*.

If you add space on the same database devices to *mydata*, and then run **sp_helpdb** again, you will 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 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 is 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 automatically 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 are adding space in order to assign a table or index to a specific segment (and, therefore, to specific devices), you will want to reduce the scope of the *system* and *default* segments to drop 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 are planning to use *new_space* to store a user table or index for improved performance, you and want to ensure that other user objects are not stored on the device by default, reduce the scope of 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 is 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 pointing to that database device, users who create tables and indexes without using the **on** clause will not be placing them on your specially prepared device.

If you use **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 will 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_pages
---------------------- -------------- -----------
newdevice              3.0MB                 1528

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

Segments and Clustered Indexes

As mentioned in "Creating Clustered Indexes on Segments" on page 16-15, if you create a table on one segment and create its clustered index on a different segment, the table travels with its index. The following example creates a clustered index, without specifying the segment name, using the same table you just created on the *new_space* segment in the example above. Checking *new_space* after the **create index** command shows that no objects remain 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_pages
------------------- ---------------- -----------
datadev2            2.0MB                   1016
```

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

17 Checking Database Consistency

This chapter describes how to check database consistency using the **dbcc** commands. It includes the following sections:

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**). "Page and Object Allocation Concepts" on page 17-3 explains page and object allocation and page linkage.

The sections that follow discuss:

- 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 **dbrepair** 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 2500–2599 or 7900–7999. 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 that have the **fix** option
- Following instructions in the *SQL Server Troubleshooting Guide*, which contains step-by-step instructions for resolving many **dbcc** errors

Whatever techniques are 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

Only the table owner can execute **dbcc** with the **checktable**, **fix_text**, and **reindex** options. Only the Database Owner can use the **checkdb**,

checkalloc, **checkcatalog**, **indexalloc**, and **tablealloc** options. You must be a System Administrator to use the **dbrepair** option.

dbcc Buffer Sizes

dbcc uses different buffer sizes, depending on the type of **dbcc** command you run and the configuration of SQL Server. The buffer sizes used are as follows:

- If you configured no named caches for SQL Server, and the **number of extent i/o buffers** configuration parameter is set to 0, all **dbcc** commands use 2K I/O buffers.
- If you configured the **number of extent i/o buffers** configuration parameter to a value of 1 or more, **dbcc checkalloc** and **dbcc tablealloc** use those extent I/O buffers, if they are available when the **dbcc** command executes. In this case, the **dbcc** commands use 1 or more 8K buffers.
- If you configured a named cache with a 16K buffer pool for one or more tables, **dbcc checkdb** and **dbcc checktable** use the 16K buffer pool when checking the consistency of those tables. Note, however, that the 16K buffer pool is used only when accessing the table itself. When checking the table's indexes, **dbcc** uses only 2K buffers.

See "number of extent i/o buffers" on page 11-85 for more information about the **number of extent i/o buffers** configuration parameter.

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. 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 are not real database objects and do not appear in system tables, but **dbcc** errors on allocation pages report this value.

When a table or index requires space, SQL Server allocates a block of 8 2K pages to the object. This 8-page block is called an **extent**. Each 256-page allocation unit contains 32 extents. SQL Server uses extents as a unit of space management to allocate and deallocate space:

- When you create a table or an 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 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.

Figure 17-1 shows how data pages are set up within extents and allocation units in SQL Server databases.

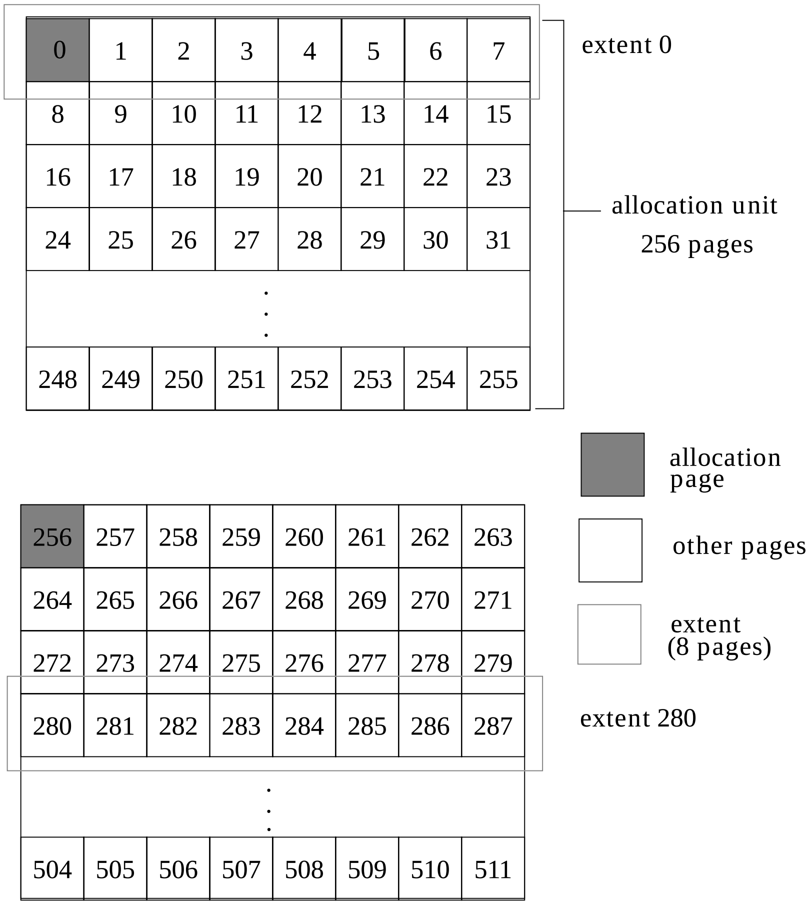


Figure 17-1: Page management with extents

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 2000 and 63,750 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 17-2 shows an object stored on 4 extents, numbered 0, 24, 272 and 504. The OAM 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.

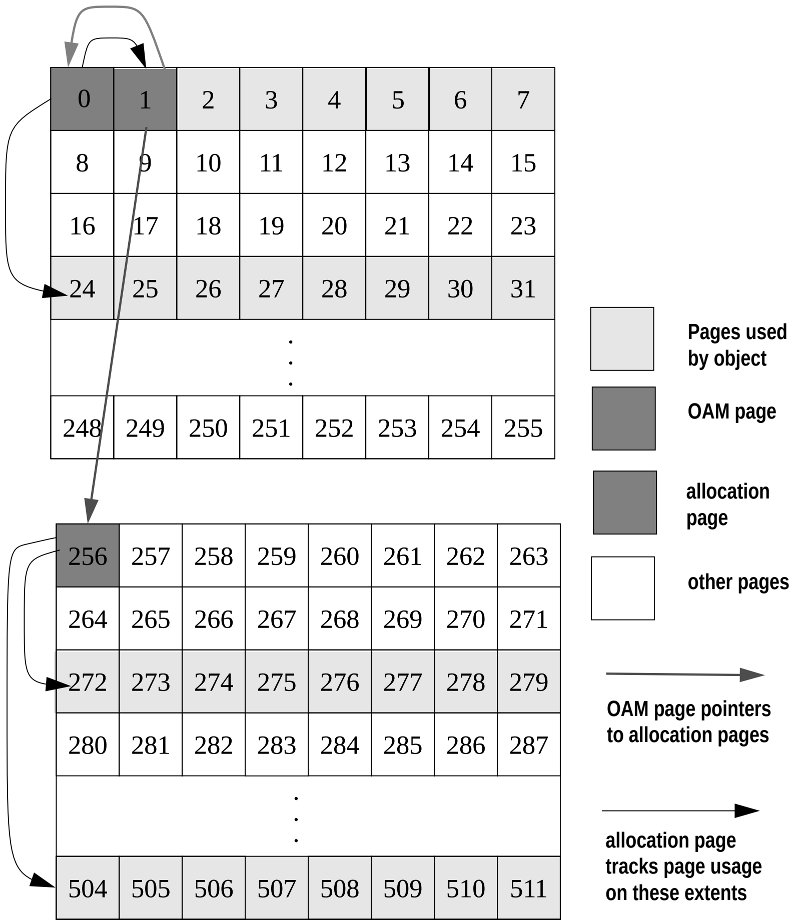


Figure 17-2: OAM page and allocation page pointers

The **dbcc checkalloc** and **dbcc tablealloc** commands examine this OAM page information, in addition to checking page linkage, as 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 17-3 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.

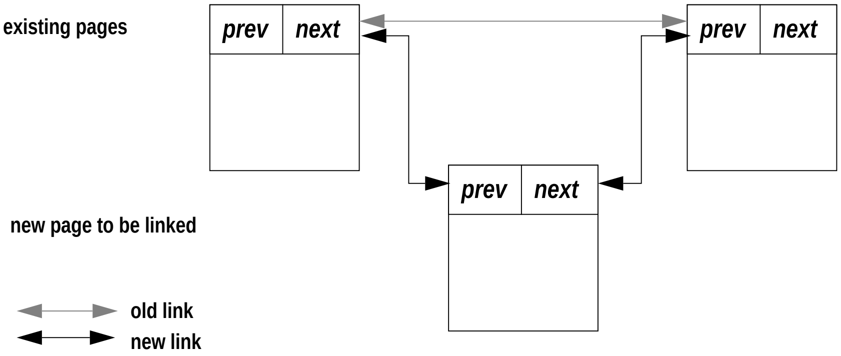


Figure 17-3: How a newly allocated page is linked with other pages

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 have entries in the first OAM page that match 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 re-create 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:

```
dbcc checktable(titles)
go

Checking titles
The total number of data pages in this table is 3.
Table has 18 data rows.

DBCC execution completed. If DBCC printed error
messages, contact a user with System Administrator
(SA) role.
```

If the table is partitioned, **dbcc checktable** checks data page linkage for each partition. For example:

```
dbcc checktable(historytab)
go

Checking historytab
The total number of pages in partition 1 is 20.
The total number of pages in partition 2 is 17.
The total number of pages in partition 3 is 19.
The total number of pages in partition 4 is 17.
The total number of pages in partition 5 is 20.
The total number of pages in partition 6 is 16.
The total number of pages in partition 7 is 19.
The total number of pages in partition 8 is 17.
The total number of pages in partition 9 is 19.
The total number of pages in partition 10 is 16.

The total number of data pages in this table is
190.
Table has 1536 data rows.

DBCC execution completed. If DBCC printed error
messages, contact a user with System Administrator
(SA) role.
```

See "Partitioning and Unpartitioning Tables" on page 13-15 in the *Performance and Tuning Guide* for more information about partitions.

To check a table that is 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 12, "Configuring Character Sets, Sort Orders, and Message Language," for more information about changing sort orders.

- If data rows are 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                102        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 issue this command only 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 that 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).*

Three types of reports 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-level errors). It does not produce an allocation report.

Table 17-1 on page 17-17 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 for all user tables is

fix; 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 the 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 12, "Configuring Character Sets, Sort Orders, and Message Language."

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 12, "Configuring Character Sets, Sort Orders, and Message Language," 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 17-1 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**.

Table 17-1: dbcc commands compared

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 the 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
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; performs 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		

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 8:00 a.m. and 5:00 p.m. or is it used heavily 24 hours a day?

  If your SQL Server is used primarily between the hours of 8:00 a.m. and 5:00 p.m., Monday through Friday, you can run **dbcc** checks at night and on weekends so that 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 run **dbcc** checks by:

  - Dumping the database to tape
  - Loading the database dumps into a separate SQL Server
  - Running **dbcc** commands on the copied database
  - Running **dbcc** commands with the **fix** options 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 run **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 that are not used in the application, thereby allowing the application access between checks:

```
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. The following is 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 the 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, contact a user
with System Administrator (SA) role.
```

Backup and Recovery

18 Developing a Backup and Recovery Plan

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 the first of a four-chapter unit on **backup** and **recovery**. It provides information to help you develop a backup and recovery plan. It includes the following topics, which provide an overview of SQL Server's backup and recovery processes:

This chapter also discusses the backup and recovery issues that you should address before you begin using your system for production, including:

Table 18-1: Further information about backup and recovery

For More Information About	See
dump, **load**, **online database**, and **sp_volchanged** syntax	Chapter 19, “Backing Up and Restoring User Databases”
Backing up and restoring the system databases	Chapter 20, “Backing Up and Restoring the System Databases”
Using thresholds to automate backups	Chapter 21, “Managing Free Space with Thresholds”

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 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 will 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 11-1 on page 11-19 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 the transactions that must be committed or rolled back.

Use the **recovery interval in minutes** configuration parameter to specify, in minutes, the maximum permissible recovery time. 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 in minutes"
```

The default value, 5, allows recovery within 5 minutes per database. Use **sp_configure "recovery interval in minutes"** to change the recovery interval. For example, to set the recovery interval to 3 minutes, issue this command:

```
sp_configure "recovery interval in minutes", 3
```

➤ ***Note***

The recovery interval has no effect on long-running, minimally-logged transactions (such as **create index**) 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. To avoid lengthy delays, dump each database immediately after creating an index on one of its tables.

Automatic Checkpoint Procedure

Approximately once each minute, the checkpoint task "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 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 task sleeps for another minute.

To see the checkpoint task, execute **sp_who**. The checkpoint task 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  HOUSEKEEPER
    5 sleeping   NULL                0   master  CHECKPOINT SLEEP
```

Checkpoint After User Database Upgrade

SQL Server inserts a checkpoint record immediately after upgrading a user database. SQL Server uses this record to ensure that a **dump database** occurs before a **dump transaction** on the upgraded database.

Truncating the Log After Automatic Checkpoints

System Administrators can use the **trunc log on chkpt** database option to truncate the transaction log when SQL Server performs an automatic checkpoint. If there are 50 or more rows in the transaction log when an automatic checkpoint occurs, SQL Server truncates the log.

To set the **trunc log on chkpt** database option, 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 set to 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 21, “Managing Free Space with Thresholds.”

Free Checkpoints

When SQL Server has no user tasks to process, a housekeeper task automatically begins writing dirty buffers to disk. If the housekeeper task is able to flush all active buffer pools in all configured caches, it wakes up the checkpoint task. The checkpoint task determines whether it can checkpoint the database.

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

For information about tuning the housekeeper task, see Chapter 17, “Using CPU Resources Effectively,”in the *Performance and Tuning Guide.*

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 **print recovery information** 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 this command:

```
sp_configure "print recovery information", 1
```

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 19, “Backing Up and Restoring User Databases” and Chapter 20, “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 17, “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 transaction** has copied the log, it truncates the inactive portion.

dump transaction takes less time and storage space than a full database backup, and it 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 **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 generated on SQL Server prior to release 10.0. If the database you are loading includes tables that contain the primary keys for tables in other databases, you must load the dump into a database with the same database name as the one dumped.

The **load database** command sets the database status to "offline." No one can use a database while it is offline, thus preventing users from making changes to the database during the database load and subsequent transaction log loads.

Effective with release 11.0, the **load database** command sets a database to "offline," so it is no longer necessary to use the **no chkpt on recovery, dbo use only**, and **read only** options of **sp_dboption** before loading a database.

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 "zero-ing" 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.

Users can not make changes to the database between the **load database** and **load tran** commands, due to the offline status set by **load database**.

You can load only the transaction log dumps that are at the same SQL Server release level as the associated database.

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.

Making the Database Available to Users: *online database*

When the load sequence completes, use the **online database** command to change the database status to "online," thus making it available to users. A database loaded by **load database** remains inaccessible until the **online database** command is issued.

Be sure you have loaded all required transaction logs before issuing the **online database** command.

The **online database** command also upgrades user databases, as described in the next section.

Moving a Database to Another SQL Server

You can use the **dump database** and **load database** commands to move a database from one SQL Server to another, as long as both SQL Servers run on the same hardware and software platform. However, you must take steps to ensure that the device allocations on the destination SQL Server match those on the original. Otherwise, system-defined and user-defined segments in the new database will not match those in the original database.

To preserve device allocations when loading a database dump in a new SQL Server, use the same instructions you would for recovering a user database from a failed device. See "Examining the Device Allocations" on page 19-40 for more information.

Upgrading a Single-User Database

You can upgrade SQL Server release 10.0 database and transaction log dumps for a single-user database to the current release of SQL Server.

➤ ***Note***

Upgrading a single-user database can occur only on databases at SQL Server release 10.0 or later.

The steps for upgrading a user database are the same as for a normal database load:

1. Use **load database** to load the most recent release 10.0 database dump. **load database** sets the database status to "offline."
2. Use **load transaction** to load, **in order**, all release 10.0 transaction logs generated after the last database dump. Be sure you have loaded all transaction logs before going to step 3.
3. Use **online database** to upgrade the database to the current version of SQL Server. When the upgrade completes, the database status is set to "online," which makes the database available for public use.
4. Use **dump database** on the newly upgraded database to create a dump that is consistent with the current release of SQL Server. A **dump database** must occur before a **dump transaction** command is permitted.

For additional information regarding the **load database**, **load transaction**, and **online database** commands in relation to upgrading a user database, see the *SQL Server Reference Manual*.

Using the Special *dump transaction* Options

In certain circumstances, the simple model described above does not apply. Table 18-2 describes when to use the special **with no_log** and **with truncate_only** options instead of the standard **dump transaction** command.

◆ ***WARNING!***

Use the special dump transaction commands only as indicated in Table 18-2. In particular, use dump transaction with no_log as a last resort and use it only once after dump transaction with no_truncate fails. The dump transaction with no_log command frees very little space in the transaction log. If you continue to load data after entering dump transaction with no_log, it is possible to fill the log completely, causing any further dump transaction commands to fail. Use the alter database command to allocate additional space to the database.

Table 18-2: When to use dump transaction with truncate_only or with no_log

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

Using the Special Load Options to Identify Dump Files

Use the **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. These options do not actually load databases or transaction logs on the tape.

➤ ***Note***

These options are mutually exclusive. If you specify both, **with listonly** prevails.

Restoring a Database from Backups

Figure 18-1 illustrates the process of restoring a database that is created at 4:30 p.m. on Monday and dumped immediately afterward. Full database dumps are made every night at 5:00 p.m.

Transaction log dumps are made at 10:00 a.m., noon, 2:00 p.m., and 4:00 p.m. every day:

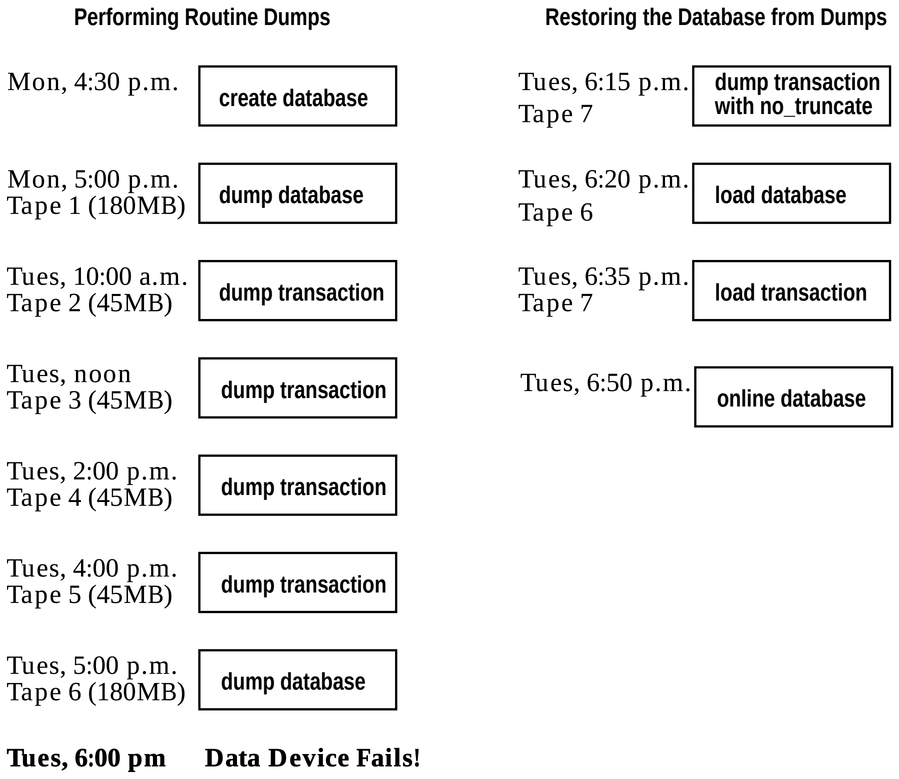


Figure 18-1: Restoring a database from backups

If the disk that stores the data fails on Tuesday at 6:00 p.m., follow these 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. **load database** sets the database status to “offline.”
3. Use **load transaction** to apply the most recent transaction log dump, Tape 7.
4. Use **online database** to set the database status to “online.”

Figure 18-2 illustrates how to restore the database when the data device fails at 4:59 p.m. on Tuesday—just before the operator is scheduled to make the nightly database dump:

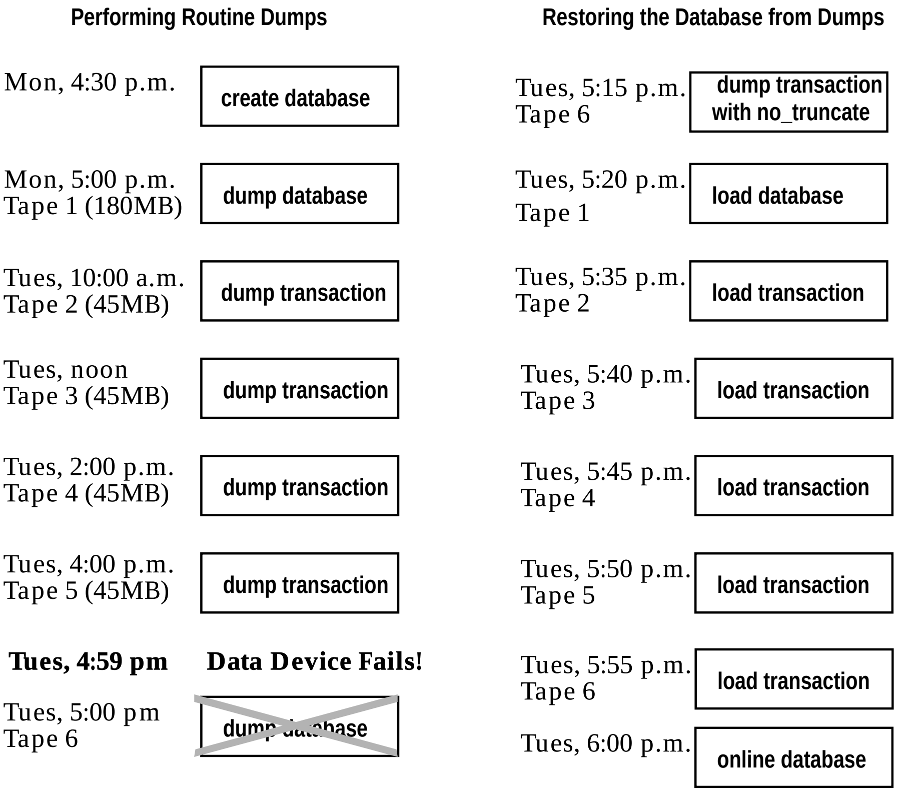


Figure 18-2: Restoring a database, a second scenario

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. **load database** sets the database status to “offline.”
3. Use **load transaction** to load Tapes 2, 3, 4, and 5 and the most recent transaction log dump, Tape 6.
4. Use **online database** to set the database status to “online.”

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. 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 18-3 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 user 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 RPCs to exchange instructions and status messages. Backup Server—not SQL Server—performs all data transfer for the dump and load commands.

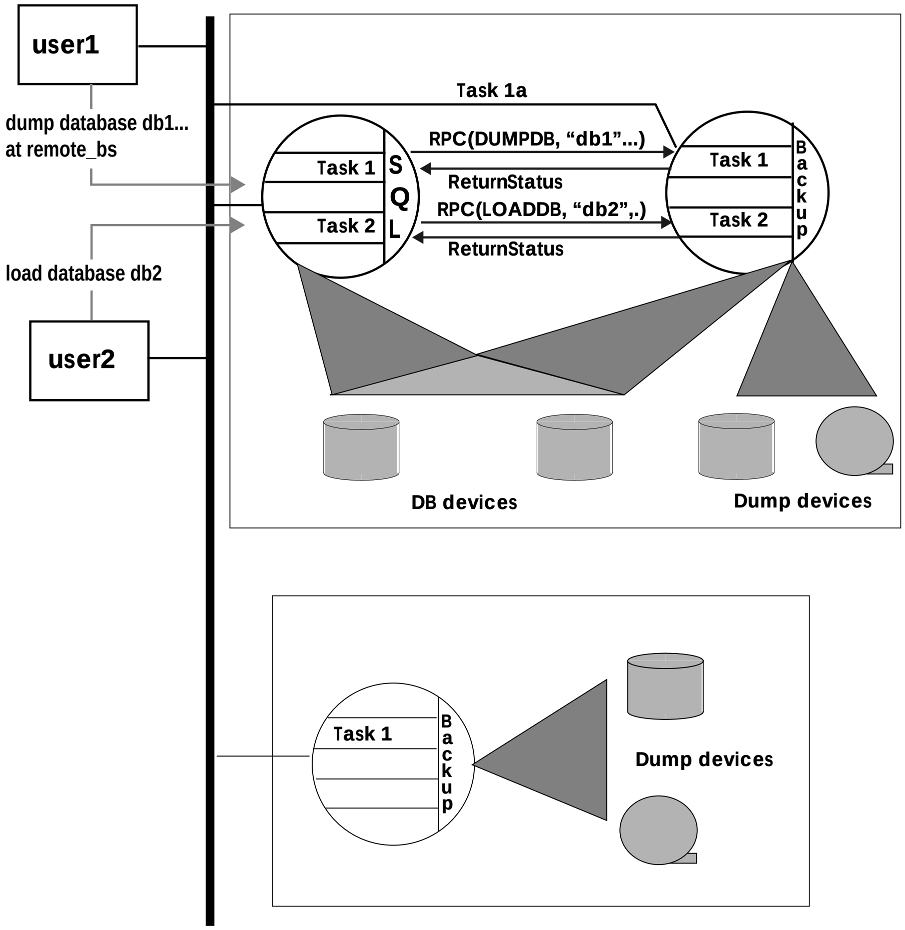


Figure 18-3: SQL Server/Backup Server installation with a remote Backup Server

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 offline 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 7, "Managing Remote Servers," in the *Security Administration Guide*.
- 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 18-20 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 and then executes **sp_volchanged**. Figure 18-3 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.

Table 18-3: Changing tape volumes on a UNIX system

Sequence	Operator, Using *isql*	SQL Server	Backup Server
1	Issues the **dump database** command		
2		Sends dump request to Backup Server	
3			Receives dump request message from SQL Server Sends message for tape mounting to operator Waits for operator’s reply
4	Receives volume change request from Backup Server Mounts tapes Executes **sp_volchanged**		
5			Checks tapes If tapes are okay, begins dump When tape is full, sends volume change request to operator
6	Receives volume change request from Backup Server Mounts tapes Executes **sp_volchanged**		
7			Continues dump When dump is complete, sends messages to operator and SQL Server
8	Receives message that dump is complete Removes and labels tapes	Receives message that dump is complete Releases locks Completes the **dump database** command	

Starting and Stopping Backup Server

Most UNIX systems use the **startserver** utility to start Backup Server on the same machine as SQL Server. See the SQL Server installation and configuration guide for information about starting SQL Server on your system.

Use the **shutdown** command to shut down a Backup Server. See Chapter 4, “Diagnosing System Problems,” and 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 "allow remote access", 0
```

Use the following commands to re-enable remote access right before a dump or load:

```
sp_configure "allow remote access", 1
```

The **allow 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 **allow remote access** variable.

Choosing Backup Media

Tapes are preferred as dump devices since they permit a library of database and transaction log dumps to be kept offline. 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 SQL Server installation and configuration guide.

Protecting Backup Tapes from Being Overwritten

The **tape retention in days** configuration parameter determines how many days' backup tapes are protected from being overwritten. When you install SQL Server, **tape retention in days** has a value of 0. This means that backup tapes can be overwritten immediately.

Use **sp_configure** to change the **tape retention in days** value. The new value takes effect the next time you reboot SQL Server:

```
sp_configure "tape retention in days", 14
```

Both the **dump database** and **dump transaction** commands provide a **retaindays** option that overrides the **tape retention in days** value for that 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, dumping to a file or disk 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 offline storage, but these tapes must be copied back to an online 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, if you are not 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 the latter 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 path name 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 you create backup scripts and threshold procedures, 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"} , logicalname,
   physicalname, tapesize
```

The *physicalname* can be either an absolute path name or a relative path name. During dumps and loads, the Backup Server resolves relative path names by looking in SQL Server's current working directory.

The *tapesize* is the capacity of the tape in megabytes. OpenVMS systems ignore the *tapesize* parameter if it is specified. Other platforms require this parameter for tape devices but ignore it for disk devices. The Backup Server uses the *tapesize* parameter if the dump command does not specify a tape capacity.

The *tapesize* must be at least 1MB, 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 upgrade a user database, create a new index, perform an unlogged operation, or run the **dump transaction with no_log** or **dump transaction with truncate_only** command.

Dumping a User Database After Upgrading

After upgrading a user database to the current version of SQL Server, you must dump the newly upgraded database to create a dump that is compatible with the current release of SQL Server. A **dump database** must occur on upgraded user databases before a **dump transaction** is permitted.

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, adding no entries (or, in the case of **bcp**, minimal entries) in the transaction log:

- Non-logged **writetext**
- **select into** on a permanent table
- Fast bulk copy (**bcp** into a table with no triggers or indexes)

Because all 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 it is necessary for you to run either command because of lack of disk space. SQL Server prohibits you from copying the transaction log until you have done so. See "Using the Special dump transaction Options" on page 18-11 for more information about **dump transaction with truncate_only** and **dump transaction with no_log** restrictions.

If the **trunc log on chkpt** database option is set to **true**, and the transaction log contains 50 or more rows, SQL Server automatically truncates the log when an automatic checkpoint occurs. Then, 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 do not have a current backup of *master*, you may have 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 or system procedures:

- **disk init**, **sp_addumpdevice** and **sp_dropdevice**
- Disk mirroring commands
- The segment system procedure **sp_addsegment, sp_dropsegment**, or **sp_extendsegment**
- **create procedure** or **drop procedure**
- **sp_logdevice**
- **sp_configure**
- **create database** or **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 re-create the changes. Otherwise, you must reissue each command against the rebuilt master database.

You should also keep a hard copy of *syslogins*. When you recover *master* from a dump, compare the hard copy to your current version of the table to be sure that users retain the same user IDs.

Truncating the *master* Database Transaction Log

Since the *master* database 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 redo all of the changes you have made in order to restore *model*.

Truncating the *model* Database'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 redo all of your changes by re-creating your procedures or re-executing your **grant** and **revoke** commands.
- Back up *sybsystemprocs* each time you change it.

Both of these recovery options are described in Chapter 20, "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. Signaling 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, read Chapter 19, "Backing Up and Restoring User Databases," and Chapter 20, "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.

19 Backing Up and Restoring User Databases

Regular and frequent backups are your only protection against database damage that results from failure of your database devices.

This chapter is the second 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 the following topics:

Table 19-1: Further information about backup and recovery

For More Information About	See
Backup and recovery issues to address before production	Chapter 18, “Developing a Backup and Recovery Plan”
Backing up and restoring the system databases	Chapter 20, “Backing Up and Restoring the System Databases”
Using thresholds to automate backups	Chapter 21, “Managing Free Space with Thresholds”

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 = *file_name*** 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 19-4 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.

Table 19-2: Syntax for routine dumps, and log dumps after device failure

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 19-3 shows the syntax for loading a database, applying transactions from the log, and returning information about dump headers and files:

Table 19-3: Syntax for the load commands

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 19-4 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

Table 19-4: Special dump transaction options

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**

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” on page 18-15.

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.

Table 19-5: Indicating the database name and 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**}]}]

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 overwrites 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 of 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 path name
  - A relative path name
  - A logical device name from the *sysdevices* system table

  The Backup Server resolves relative path names using SQL Server's current working directory.
- When dumping or loading across the network:
  - You must specify the absolute path name of the dump device. (You cannot use a relative path name or a logical device name from the *sysdevices* system table.)
  - The path name must be valid on the machine on which the Backup Server is running.
  - If the name includes any characters 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:"
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

Tape Device Determination by Backup Server

To become less device dependent and more flexible, Backup Server provides a method for determining the tape device characteristics (tape positioning for read, close, append, I/O size, file marks, and ability to overwrite a tape mark) for a dump operation.

When you issue a **dump database** or **dump transaction** command, Backup Server checks to see if the device type of the specified dump device is known (supplied and supported internally) by SQL Server. If the device is not a known type, Backup Server checks the tape configuration file (default location is *$SYBASE/backup_tape.cfg)* for the device configuration. If the configuration is found, the **dump** command proceeds.

If the configuration is not found in the tape device configuration file, the **dump** command fails with the following error message:

```
Device not found in configuration file. INIT needs
to be specified to configure the device.
```

This means the device needs to be configured. Issue the **dump database** or **dump transaction** with **init** qualifier to configure the device. Using operating system calls, Backup Server attempts to determine the device's characteristics, and if successful, stores the device characteristics in the tape configuration file.

If Backup Server cannot determine the dump device characteristics, it defaults to one dump per tape. The device can not be used if the configuration fails to write at least one dump file.

Tape configuration by Backup Server applies only to Unix platforms.

Tape Device Configuration File

Format

The tape device configuration file contains tape device information that is used only by the **dump** command.

The format of the file is one tape device entry per line. Fields are separated by blanks or tabs.

Creation

This file is not provided at installation or upgrade time. It is created only when Backup Server is ready to write to it (**dump database** or **dump transaction** with **init**). When Backup Server tries to write to this file for the first time, the following warning message is issued:

```
Warning, unable to open device configuration file
for reading. Operating system error. No such file
or directory.
```

Ignore this message. Backup Server gives this warning and then creates the file and writes the configuration information to it.

Manual Editing

Since Backup Server updates the configuration file, the only user interaction with the file occurs when the user receives the following error message:

```
Device does not match the current configuration.
Please reconfigure this tape device by removing
the configuration file entry and issuing a dump
with the INIT qualifier.
```

This means the tape hardware configuration changed for a device name. Delete the line entry for that device name and issue a **dump** command as instructed.

Default Location

The default path name for the configuration file is *$SYBASE/backup_tape.cfg*. To change the default location, use the Backup Server Configuration menu on **sybinit**.

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.

Table 19-6: Dumping to or loading from a remote Backup Server

	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**}]}]

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
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

Specifying Tape Density, Block Size, 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.

Table 19-7: Specifying tape density, blocksize, and capacity

	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**}]}]

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" on page 19-22.

Overriding the Default Block Size

By default, the dump and load commands choose the "best" block size for your operating system. Wherever possible, use these defaults.

You can use the **blocksize** = ***number_bytes*** option to override the default block size for a particular dump device. The block size must be at least one database page (2048 bytes), and must be an exact multiple of the database page size.

For OpenVMS systems, you can specify a block size only for dumps. Use a block size less than or equal to 55,296.

For UNIX systems, you can specify a block size on both dump and load commands. When loading a dump, you must specify the same block size 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, **and** 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 path name 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). We recommend that you specify a capacity that is slightly below the rated capacity for your device.

A rule of thumb for calculating capacity is to use 70 percent of the manufacturer's maximum capacity for the device, and allow 30 percent for overhead (inter-record gaps, tape marks, and so on). This rule of thumb works in most cases, but may not work in all cases because of differences in overhead across vendors and across devices.

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.

Table 19-8: Specifying the volume name

	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***,

Table 19-8: Specifying the volume name (continued)

	Backing Up a Database or Log	Loading a Database or Log
	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**}]}]

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 7 characters of the database name
- 2-digit year number
- 3-digit day of the year (1–366)
- 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.

Table 19-9: Specifying the file name for a dump

	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***]

Table 19-9: Specifying the file name for a dump (continued)

	Backing Up a Database or Log	Loading a Database or Log
	[**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**}]}]

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:

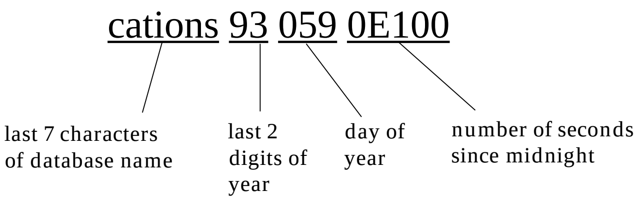

cations 93 059 0E100
last 7 characters of database name
last 2 digits of year
day of year
number of seconds since midnight

Figure 19-1: 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 p.m.) 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"
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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):

Table 19-10:Using more than one dump device

	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}]}]

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:"
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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

Table 19-11:Tape handling options

	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**}]}]

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 in days** configuration parameter 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 in days** parameter 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 in days and **retaindays** are meaningful only for disk, 1/4-inch cartridge, and single-file media. On multi-file media, the Backup Server checks only 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
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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 19-2 illustrates which options to use to dump three databases to a single tape volume.

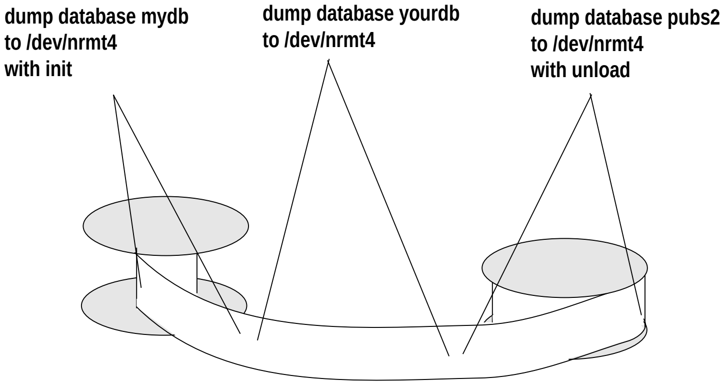


Figure 19-2: Dumping several databases to the same volume

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 19-12:Overriding the default message destination

	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,
	[nodismount | dismount],
	[nounload | unload],
	retaindays = number_days,
	[noinit | init],
		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],

Message destination	[notify = {client | operator_console}]}]	[notify = {client | operator_console}]}]

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 that information.

Table 19-13:Listing dump headers or file names

	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**}]}]

➤ ***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"
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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.

The **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.

Copying the log **with no_truncate** is the first step described in "Recovering a Database: Step-by-Step Instructions" on page 19-39. To completely recover a database, follow the steps in that section.

Table 19-14:Copying the log after a device failure

	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***, **dump volume =** ***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**}]}]

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, and 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 re-creating 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 used **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 as a last resort, and use it only once after dump transaction with truncate_only fails. The dump transaction with no_log command frees very little space in the transaction log. If you continue to load data after entering dump transaction with no_log, it is possible to fill the log completely, causing any further dump transaction commands to fail. Use the alter database command to allocate additional space to the database.

All occurrences of **dump tran with no_log** are reported in the SQL Server error log. The message includes the user ID of the user executing the command. Messages indicating success or failure are also sent to the error log. **no_log** is the only dump option that generates error log messages.

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 to ...
```

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.

However, some situations can still cause the transaction log to become too full, even with frequent log dumps. The **dump transaction** command truncates the 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 (known as the oldest active transaction). The longer this active transaction remains uncommitted, the less space is available in the transaction log, since **dump transaction** cannot truncate additional pages.

This can happen in situations when applications with very long transactions modify tables in a database with a small transaction log, which indicates you should increase the size of the log. It also occurs when transactions inadvertently remain uncommitted for long periods of time such as when an implicit **begin transaction** uses the chained transaction mode or when a user forgets to complete the transaction. You can determine the oldest active transaction in each database by querying the *syslogshold* system table.

The *syslogshold* Table

The *syslogshold* table exists in the *master* database. Each row in the table represents either:

- The oldest active transaction in a database, or
- The Replication Server® truncation point for the database’s log.

A database may have no rows in *syslogshold*, a row representing one of the above, or two rows representing both of the above. For information about how a Replication Sever truncation point affects the truncation of the database’s transaction log, see your Replication Server documentation.

Querying *syslogshold* provides a “snapshot” of the current situation in each database. Since most transactions last for only a short time, the query’s results may not be consistent. For example, the oldest active transaction described in the first row of *syslogshold* may finish before SQL Server completes the query of *syslogshold.* However, when several queries of *syslogshold* over time result in the same row for a database, that transaction may prevent a **dump transaction** from truncating any log space.

Once the transaction log reaches the last-chance threshold and **dump transaction** cannot free up space in the log, you can query *syslogshold* and *sysindexes* to identify the transaction holding up the truncation. For example:

```
select H.spid, H.name
from master..syslogshold H, threshdb..sysindexes I
where H.dbid = db_id("threshdb")
and I.id = 8
and H.page = I.first
```

```
spid    name
------  ------------------------------------
     8  $user_transaction

(1 row affected)
```

This query uses the object ID associated with *syslogs* (8) in the *threshdb* database to match the first page of its transaction log with the first page of the oldest active transaction in *syslogshold*.

You can also query *syslogshold* and *sysprocesses* in the *master* database to identify the specific host and application owning the oldest active transactions. For example:

```
select P.hostname, P.hostprocess, P.program_name,
   H.name, H.starttime
from sysprocesses P, syslogshold H
where P.spid = H.spid
and H.spid != 0
```

```
hostname  hostprocess  program_name  name

--------  -----------  ------------  ----------------------
    starttime
    ----------------------------
eagle     15826       isql          $user_transaction
    Sep  6 1995  4:29PM
hawk      15859       isql          $user_transaction
    Sep  6 1995  5:00PM
condor    15866       isql          $user_transaction
    Sep  6 1995  5:08PM

(3 rows affected)
```

Using the above information, you can notify or kill the user process owning the oldest active transaction and proceed with the **dump transaction**. You can also include the above types of query in the threshold procedures for the database as an automatic alert mechanism. For example, you may decide the transaction log should never reach its last-chance threshold. If it does, your last-chance threshold procedure (**sp_thresholdaction**) alerts you with information about the oldest active transaction preventing the transaction dump.

➤ ***Note***

The initial log records for a transaction may reside in a user log cache, which is not visible in *syslogshold* until the records are flushed to the log (for example, after a checkpoint).

For more information about the *syslogshold* system table, see the *SQL Server Reference Supplement*. For information about the last-chance threshold and threshold procedures, see Chapter 21, "Managing Free Space with Thresholds."

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, devname , action
   [ ,fname [, vname] ]
```

- Use the *session_id* and *devname* parameters specified in the volume change request.
- *action* specifies whether to **abort**, **proceed** with, or **retry** the dump or load.
- *fname* specifies which file to load. If you do not specify a file name with **sp_volchanged**, the Backup Server loads the **file** = ***file_name*** 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 *vname* 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 *vname* to confirm that the correct tape has been mounted. If you do not specify a *vname* 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***, ***devname***, **abort**
Mount a new volume and proceed with the dump	**sp_volchanged** ***session_id***, ***devname***, **proceed** [, ***fname*** [, ***vname***]]

- `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***, ***devname***, **abort**
Mount the next volume and proceed with the dump	**sp_volchanged** ***session_id***, ***devname***, **proceed** [, ***fname*** [, ***vname***]]

- `Volume on device devname 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*, *devname*, abort**
Mount another volume and retry the dump	**sp_volchanged *session_id*, *devname*, retry** [, ***fname*** [, ***vname***]]
Proceed with the dump, overwriting any existing contents	**sp_volchanged *session_id*, *devname*, proceed** [, ***fname*** [, ***vname***]]

- `Volume on device devname 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*, *devname*, abort**
Mount another volume and retry the dump	**sp_volchanged *session_id*, *session_id*, retry** [, ***session_id*** [, ***session_id***]]
Proceed with the dump, overwriting any existing contents	**sp_volchanged *session_id*, *session_id*, proceed** [, ***session_id*** [, ***session_id***]]

- `Volume to be overwritten on 'devname' 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*, *session_id*, abort**

The operator can	By replying
Mount another volume and retry the dump	**sp_volchanged *session_id*, *session_id*, retry** [, ***session_id*** [, ***session_id***]]
Proceed with the dump, overwriting any existing contents	**sp_volchanged *session_id*, *session_id*, proceed** [, ***session_id*** [, ***session_id***]]

- `Volume to be overwritten on 'devname' 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*, *session_id*, abort**
Mount another volume and retry the dump	**sp_volchanged *session_id*, *session_id*, retry** [, ***session_id*** [, ***session_id***]]
Proceed with the dump, overwriting any existing contents	**sp_volchanged *session_id*, *session_id*, proceed** [, ***session_id*** [, ***session_id***]]

Volume Change Prompts for Loads

Following are the volume change prompts and possible operator actions during loads:

- `Dumpfile 'fname' section vname found instead of 'fname' section vname.`

  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*, *session_id*, abort**
Mount another volume and try to load it	**sp_volchanged *session_id*, *session_id*, retry** [, ***session_id*** [, ***session_id***]]

The operator can	By replying
Load the file on the currently mounted volume, even though it is not the specified file (not recommended)	**sp_volchanged** ***session_id*****,** ***session_id*****, proceed** [, ***session_id*** [, ***session_id***]]

- `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*****,** ***session_id*****, abort**
Mount the next volume and proceed with the load	**sp_volchanged** ***session_id*****,** ***session_id*****, proceed** [, ***session_id*** [, ***session_id***]]

- `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*****,** ***session_id*****, abort**
Mount another volume and proceed with the load	**sp_volchanged** ***session_id*****,** ***session_id*****, proceed** [, ***session_id*** [, ***session_id***]]

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. Re-create the databases, one at a time.
7. Load the most recent database dump into each database.
8. Apply each transaction log dump in the order it was created.

These steps are described in 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 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 re-creating your databases to ensure that your log, data, and indexes reside on separate devices, and to preserve the scope of any user segments you have created.

➤ ***Note***

You can also use these steps to preserve segment mappings when moving a database dump from one server to another (on the same hardware and software platform).

If you do not use this information to re-create the device allocations for damaged databases, SQL Server will **remap** the *sysusages* table after **load database** to account for discrepancies. This means that the database's system-defined and user-defined segments no longer match the appropriate device allocations. Incorrect information in *sysusages* can result in the log being stored on the same devices as the data, even if data and log were separate before recovery. It can also change user-defined segments in unpredictable ways, and can result in a database that could not be created using a standard **create database** command.

Follow these steps to examine and record the device allocations for all damaged databases:

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" represents a log allocation. Higher values indicate user-defined segments; treat these as data allocations to preserve the scope of these segments. The *size* column indicates the number of 2K blocks of data. To find the number of megabytes, divide by 512. Note the order, use, and size of each disk piece.

   For example, this output:

```
segmap        size
-------       --------
     3          10240
     3           5120
     4           5120
     8           1024
     4           2048
```

translates into the sizes and uses described in Table 19-15.

Table 19-15:Sample device allocation

Device Allocation	Megabytes
Data	20
Data	10
Log	10
Data (user-defined segment)	2
Log	4

➤ ***Note***

If the *segmap* column contains 7’s, your data and log are on the same device, and you can recover only 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 the preceding 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 *SQL Server Troubleshooting Guide* for more information about **dbcc dbrepair.**

Dropping the Failed Devices

After you have dropped each database, use **sp_dropdevice** to drop the failed device. See the *SQL Server Reference Manual* for information on this system procedure.

Initializing New Devices

Use **disk init** to initialize the new database devices. See Chapter 6, “Initializing Database Devices,” for complete information.

Re-Creating the Databases

Use the following steps to re-create 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 that is at least as large as the original.

1. Use the **create database** command with the **for load** option. Duplicate all device fragment 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 re-create the *mydb* database allocations shown in Table 19-15 on page 19-41, enter the command:

```
create database mydb
    on datadev1 = 20,
       datadev2 = 10
log on logdev1 = 10
for load
```

➤ ***Note***

create database...for load temporarily locks users out of the newly created database, and **load database** marks the database offline for general use. This prevents users from performing logged transactions during recovery.

2. Use the **alter database** command with the **for load** option to re-create the remaining entries, in order. Remember to treat device allocations for user segments as you would data allocations.

   In this example, to allocate more data space on *datadev3* and more log space on *logdev1*, the command is:

   ```
   alter database mydb

        on datadev3 = 2
   log on logdev1=4
   for load
   ```

Loading the Database

Reload the database using **load database**. If the original database stored objects on user-defined segments (*sysusages* reports a *segmap* greater than 4) and your new your new device allocations match those of the dumped database, SQL Server preserves the user segment mappings.

If you did not create the new device allocations to match those of the dumped database, SQL Server will remap segments to the available device allocations. This remapping may also mix log and data on the same physical device.

➤ ***Note***

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.

Bringing the Databases Online

After you have applied all transaction log dumps to a database, use the **online database** command to make it available for use. In this example, the command to bring the *mydb* database online is:

```
online database mydb
```

Replicated Databases

Replicated databases must not be brought online until the logs are drained. If you try to bring a replicated database online before the logs are drained, SQL Server issues the following message:

```
Database is replicated, but the log is not yet
drained. This database will come online
automatically after the log is drained.
```

When Replication Server, via LTM, drains the log, the **online database** command is automatically issued.

Upgrading to Release 11.0

Before upgrading replicated databases to release 11.0, the databases must be online. Replicated databases will not come online until after their logs are drained (which will automatically issue the **online database** command).

Refer to the SQL Server installation and configuration guide for upgrade instructions for SQL Server users that have replicated databases.

Load Sequence

The load sequence for loading replicated databases is: **load database**, **replicate**, **load transaction**, **replicate**, and so on. At the end of the load sequence, you must issue the **online database** command to bring the databases online. Databases that are offline because they are in a load sequence are not automatically brought online by Replication Server. It is crucial that you do not issue the **online database** command until all transaction logs are loaded.

Upgrading User Database Dumps

When a SQL Server installation is upgraded to a new release, all databases associated with that server are automatically upgraded.

As a result, database and transaction log dumps created with any previous SQL Server must be upgraded before they can be used with the current SQL Server.

SQL Server release 11.0 provides an automatic upgrade mechanism, on a per-database basis, for upgrading a database or transaction log dump from any SQL Server release 10.0 to the current SQL Server release, thus making the dump compatible for use. This mechanism is entirely internal to SQL Server release 11.0 and requires no external programs. It provides the flexibility of upgrading individual dumps as needed.

The following tasks are not supported by this automatic upgrade functionality:

- Loading a SQL Server release 10.0 *master* database onto SQL Server release 11.0.
- Installing new or modified stored procedures. Continue to use **installmaster**.
- Loading and upgrading dumps prior to SQL Server release 10.0.

How to Upgrade a Dump to SQL Server Release 11.0

The process of upgrading a user database or transaction log dump to the current release of SQL Server takes the following steps:

1. Use **load database** and **load transaction** to load the dump to be upgraded.

   SQL Server determines from the dump header which version it is loading. After the dump header is read, and before Backup Server begins the load, the database is marked offline by the **load database** or **load transaction** commands. This makes the database unavailable for general use (queries and **use database** are not permitted), provides the user greater control over load sequences, and eliminates the possibility that other users will accidentally interrupt a load sequence.

2. Use the **online database** command, after the dump has successfully loaded, to activate the upgrade process.

Do **not** issue the **online database** command until after all transaction dumps are loaded.

Prior to SQL Server release 11.0, a database was automatically available at the end of a successful load sequence. With SQL Server release 11.0, the user is required to bring the database online after a successful load sequence, using the **online database** command.

For dumps loaded from SQL Server release 10.0, the **online database** command activates the upgrade process to upgrade the dumps just loaded. After the upgrade is successfully completed, SQL Server places the database online and the database is ready for use.

For dumps loaded from SQL Server release 11.0, no upgrade process is activated. You must still issue the **online database** command to place the database online, making it available for public use. (The **load database** command marks the database offline.)

During the upgrade process, each upgrade step produces a message stating what it is about to do.

An upgrade failure leaves the database offline and produces a message stating that the upgrade failed and the user must correct the failure.

For more information about **online database**, see the *SQL Server Reference Manual*.

3. After successful execution of the **online database** command, use the **dump database** command. The database must be dumped before a **dump transaction** is permitted. A **dump transaction** on a newly created or upgraded database is not permitted until a successful **dump database** has occurred.

The "Database Offline" Status Bit

The "database offline" status bit indicates that the database is not available for general use. You can determine if a database is offline by using **sp_helpdb**. It will show that the database is offline if this bit is set.

When a database is marked offline by **load database**, a status bit in the *sysdatabases* table is set and remains set until the successful completion of **online database**.

The *database offline* status bit works in combination with any existing status bits. It augments the following status bit to provide additional control:

- In recovery

The *database offline* status bit overrides the following status bits:

- DBO use only
- Read only

The following status bits override the *database offline* status bit:

- Began upgrade
- Bypass recovery
- In load
- Not recovered
- Suspect
- Use not recovered

Although the database is not available for general use, the following commands are permitted when the database is offline:

- **dump database** and **dump transaction**
- **load database** and **load transaction**
- **alter database on device**
- **drop database**
- **online database**
- **dbcc** diagnostics (subject to **dbcc** restrictions)

Version Identifiers

The automatic upgrade feature provides version identifiers for SQL Server, databases, and log record formats. They are:

Configuration Upgrade Version ID

This identifier shows the current version of SQL Server and is stored in the *sysconfigures* system table. **sp_configure** displays the current version of SQL Server as “upgrade version.”

Upgrade Version Indicator

This identifier shows the current version of a database and is stored in the database and dump headers. The SQL Server recovery mechanism uses this value to determine whether the database should be upgraded before being made available for general use.

Values are:

- 0 for SQL Server release 10.0
- 1 for SQL Server release 11.0

Log Compatibility Version Specifier

This identifier differentiates System 10™ logs from System 11™ logs by showing the format of log records in a database, database dump, or transaction log dump. This constant is stored in the database and dump headers and is used by SQL Server to detect the format of log records during recovery.

Cache Bindings and Loading Databases

You need to be aware of cache bindings for a database and the objects in the database if you dump a database and load it onto a server with different cache bindings. You may want to load the database onto a different server for tuning or development work, or you may need to load a database that you dropped from a server whose cache bindings have changed since the dump was made.

When a database is brought online after recovery or with the **online database** command after a load, SQL Server verifies all cache bindings for the database and database objects. If a cache does not exist, SQL Server writes a warning to the error log, and the binding in *sysattributes* is marked as invalid. Here is an example of the message from the errorlog:

```
Cache binding for database '5', object
'208003772', index '3' is being marked invalid in
Sysattributes.
```

Invalid cache bindings are not deleted. If you create a cache of the same name and restart SQL Server, the binding is marked as valid and the cache is used. If you do not create a cache with the same name, you can bind the object to another cache or allow it to use the default cache.

In the following sections, which discuss cache binding topics, “destination server” refers to the server where the database is being loaded, and “original server” refers to the server where the dump was made.

If possible, re-create caches that have the same names on the destination server as bindings on the original server. You may want to configure pools in exactly the same manner if you are using the destination database for similar purposes or for performance testing and development that may be ported back to the original server. If you are using the destination database for decision support or for running **dbcc** commands, you may want to configure pools to allow more space in 16K memory pools.

Databases and Cache Bindings

Binding information for databases is stored in *master..sysattributes*. No information about database binding is stored in the database itself. If you use **load database** to load the dump over an existing database that is bound to a cache, and you do not drop the database before you issue the load command, this does not affect the binding.

If the database that you are loading was bound to a cache on the original server, your options are:

- Bind the database on the destination server to a cache configured for the needs on that server.
- Configure pools in the default data cache on the destination server for the needs of the application there, and do not bind the database to a named data cache.

Database Objects and Cache Bindings

Binding information for objects is stored in the *sysattributes* table in the database itself. If you frequently load the database onto the destination server, the simplest solution is to configure caches of the same name on the destination server.

If the destination server is not configured with caches of the same name as the original server, bind the objects to the appropriate caches on the destination server after you bring the database online, or be sure that the default cache is configured for your needs on that server.

Checking on Cache Bindings

The **sp_helpcache** system procedure displays the cache bindings for database objects, even if the cache bindings are invalid.

The following SQL statements reproduce cache binding commands from the information in a user database's *sysattributes* table:

```
/* create a bindcache statement for tables */

select "sp_bindcache "+ char_value + ", "
    + db_name() + ", " + object_name(object)
from sysattributes
where class = 3
    and object_type = "T"
```

```
/* create a bindcache statement for indexes */

select "sp_bindcache "+ char_value + ", "
    + db_name() + ", "   + i.name
from sysattributes, sysindexes i
where class = 3
    and object_type = "I"
    and i.indid = convert(tinyint, object_info1)
    and i.id = object
```

20 Backing Up and Restoring the System Databases

This chapter is the third of a four-chapter unit on backup and recovery. It explains how to restore the *master*, *model* and *sybsystemprocs* databases. It contains the following topics:

Table 20-1: Further information about backup and recovery

For More Information About	See
Backup and recovery issues to address before production	Chapter 18, "Developing a Backup and Recovery Plan"
dump, **load**, and **sp_volchanged** syntax	Chapter 19, "Backing Up and Restoring User Databases"
Using thresholds to automate backups	Chapter 21, "Managing Free Space with Thresholds"

Introduction

The recovery procedure for system databases is different depending on the particular database involved and the problems that you have on your system. In general, the 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

Follow the instructions in this chapter carefully to understand the correct recovery procedure to use.

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 on 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. Shut down SQL Server and 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 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 and shutdown SQL Server.

12. Check for consistency: compare your hard copy of *sysusages* and *sysdatabases* with the new online 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 about 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.

◆ ***WARNING!***

Shut down SQL Server before you use any buildmaster command. If you use buildmaster on a master device that is in use by SQL Server, the recovery procedure will fail when you attempt to load the most recent backup of *master*.

Rebuilding Only the *master* Database

Run **buildmaster -m** (UNIX and PC) 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 as large as or larger than the size originally used to configure SQL Server. If you recorded this information in your SQL Server installation and configuration 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 2K 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
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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.

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 PC) or **/masterrecover** (OpenVMS) options. See the SQL Server utility programs manual for complete syntax.

On UNIX platforms:

```
startserver -f RUN_server_name -m
```

On OpenVMS:

```
startserver /server = server_name /masterrecover
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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. For example the database does not know about any of your database devices. 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 Device 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 20-12.

◆ ***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.

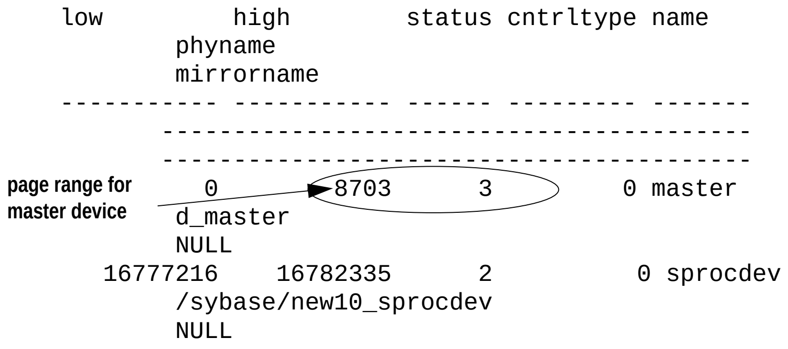
If you have more than one row for *dbid* 1 in your hard copy of *sysusages*, you need to increase the size of *master* so that you can load the dump. You must duplicate the *vstart* value for each allocation for *master* in *sysusages*. This is easiest to do if you have a copy of *sysusages* ordered 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 are not included in this sample):



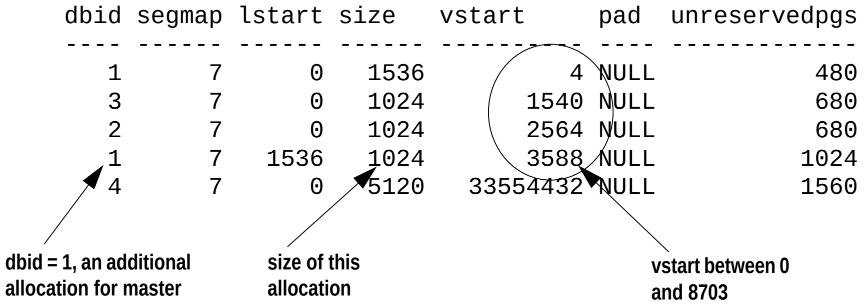
```
low         high        status cntrltype name
        phyname
        mirrorname
----------- ---------- ------ --------- -------
       ----------------------------------------
       ----------------------------------------
          0        8703      3        0 master
        d_master
        NULL
   16777216   16782335      2          0 sprocdev
        /sybase/new10_sprocdev
        NULL
```

Figure 20-1: Determining allocations on the 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
```

Figure 20-2: Sample output from sysusages

In this 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 hard copy.

The fourth row shows an additional allocation for *master*, with *vstart* = 3588 and *size* = 1024.

Figure 20-3 shows the storage allocations for the above *sysusages* data.

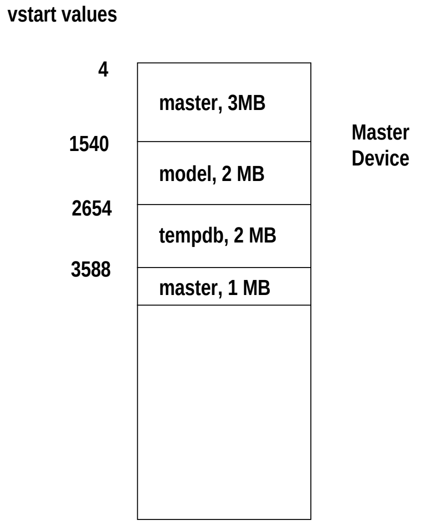


Figure 20-3: Allocations on a master device

In this 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. 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 into the server as "sa". Remember that **buildmaster** has set the password for this account to NULL. Issue the **alter database** command. For the example above, use:

```
alter database on master = 2
```

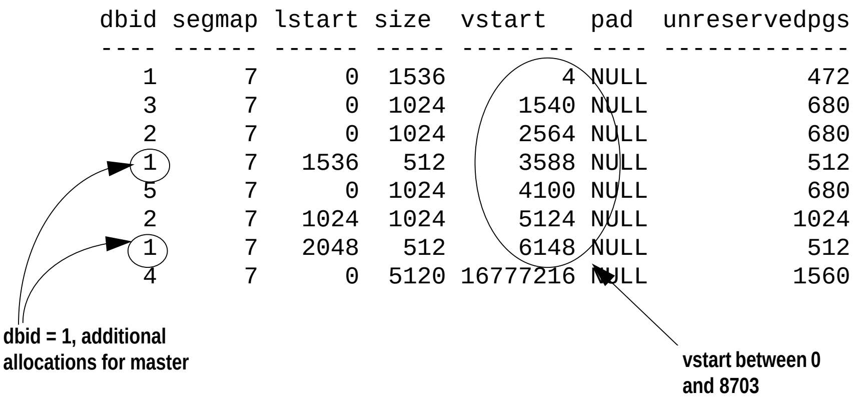
Check the *size* and *vstart* values for the new row in *sysusages*.

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

Figure 20-4: Sample sysusages output with additional allocations

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 of *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 20-5 shows the allocations on the master device.

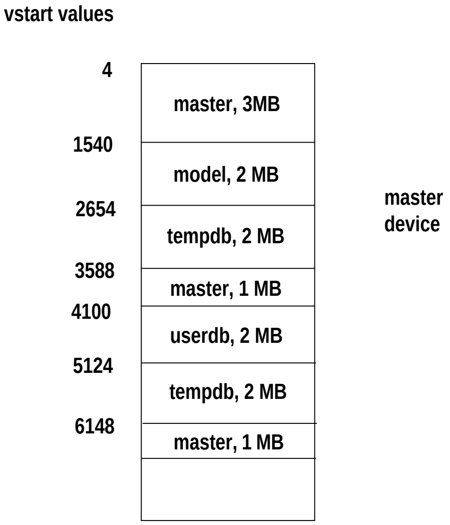


Figure 20-5: Complex allocations on a master device

You need to issue a set of **alter database** and **create database** commands to re-create all of the allocations. If your *sysusages* table lists additional allocations on the master device after the last allocation for master, you do not have to re-create 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.

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 database 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 a user database that is stored on the master device. If the user database has not been backed up since it was last changed, do not proceed. Technical Support may be able to recover it 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 20-4.
- If your *sysusages* values look correct, proceed to the next step.

Checking Your Backup Server *sysservers* Information

Log into the server as “sa”, using a 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 be sure 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

On UNIX and OpenVMS platforms, 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 PC platforms, see the SQL Server installation and configuration guide for information on how to verify that your Backup Server is running, and see the *SQL Server Utility Programs* manual for the commands to start Backup Server.

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 19, "Backing Up and Restoring User Databases," for information on command syntax. See the SQL Server installation and configuration guide 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 hard copy. Look especially for these problems:

- If any devices in your hard copy 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 any databases listed in your hard copy 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 one exists. 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.

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 (including the original value of **vstart**). If the size you provide is too small, or if you use a different **vstart** value, you may corrupt your database.

If you do not save your **disk init** scripts, look at your most recent hard copy of *sysdevices* to determine some of the correct parameters for **disk reinit**. You will still need to know the value of **vstart** if you used a custom **vstart** in the original **disk init** command.

Table 20-2: Using sysdevices to determine disk reinit parameters

disk reinit parameter	*sysdevices* data	Notes
name	*name*	Use the same name, especially if you have any scripts that 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 be sure to use a value not already in use.
size	(*high*-*low*) +1	Extremely important to provide correct size information.

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 be run only 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 6, “Initializing Database Devices,” or see the *SQL Server Reference Manual*.

Rebuilding *sysusages* and *sysdatabase*

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
```

SQL Server automatically shuts down after **disk refit** rebuilds the system tables. Examine the output while **disk refit** runs and during shutdown to determine if any errors occurred.

◆ ***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 online version.
2. Compare your hard copy of *sysdatabases* with the new online version.
3. Run **dbcc checkalloc** on each database.
4. Examine the important tables in each database.

◆ ***WARNING!***

If you find discrepancies in *sysusages*, call Sybase 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
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

- Redo any changes you have made to *model*.

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 hard copy 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*.

◆ ***WARNING!***

Shut down SQL Server before you use any buildmaster command.

On UNIX platforms:

```
buildmaster -d /devname -x
```

On OpenVMS:

```
buildmaster /disk = physicalname /model
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

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, reissue 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:

- Follow the steps above for restoring a generic model database
- Reissue all 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
```

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 6, “Initializing Database Devices,” 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 = 16
```

➤ ***Note***

The required size for the *sybsystemprocs* database may be different for your operating system. See the SQL Server installation and configuration guide for the correct size.

4. Run the **installmaster** script.

◆ ***WARNING!***

Do not run the *installmaster* script repeatedly without first dropping and re-creating the *sybsystemprocs* database. Running *installmaster* repeatedly can change the distribution of index values in such a way that the *sysprocedures* table will require much more disk space to store the same amount of data. To avoid this problem, drop and re-create the *sybsystemprocs* database before running *installmaster*.

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
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

5. If you have made any changes to permissions in *sybsystemprocs*, or if you have added your own procedures to the database, you must redo 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 reissuing 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, be sure that the complete backup fits on one tape volume or that more than one SQL Server is able to communicate with your Backup Server. If a dump spans more than one tape volume, issue the change-of-volume command using the system procedure **sp_volchanged**, which is stored in *sybsystemprocs*. You cannot issue that command in the middle of recovering a database.

Following are sample load commands:

On UNIX:

```
load database sybsystemprocs from "/dev/nrmt4"
```

On OpenVMS:

```
load database sybsytemprocs from "MTA0:"
```

On PC platforms:

For PC platform examples, refer to the SQL Server installation and configuration guide.

21 Managing Free Space with Thresholds

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 the last 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 includes the following topics:

Table 21-1: Further information about backup and recovery

For More Information About	See
Backup and recovery issues to address before production	Chapter 18, "Developing a Backup and Recovery Plan"
dump, **load**, and **sp_volchanged** syntax	Chapter 19, "Backing Up and Restoring User Databases"
Backing up and restoring the system databases	Chapter 20, "Backing Up and Restoring the System Databases"

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 21-1 illustrates a log segment with a last-chance threshold. The shaded area represents log space that has already been used; the unshaded area represents free log space. The last-chance threshold has not yet been crossed.

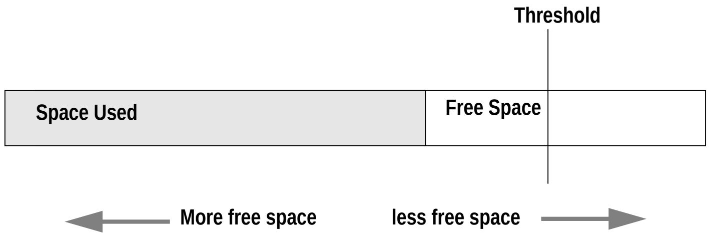


Figure 21-1: Log segment with a last-chance threshold

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_thresholdaction**:

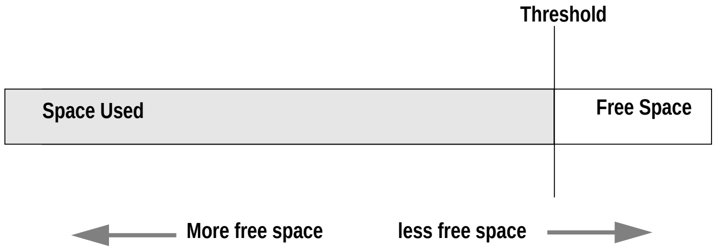


Figure 21-2: Executing sp_thresholdaction when the last-chance threshold is reached

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 21-2 executes **sp_thresholdaction**, it is deactivated. In Figure 21-3, the threshold is reactivated when the amount of free space increases by *@@thresh_hysteresis* pages:

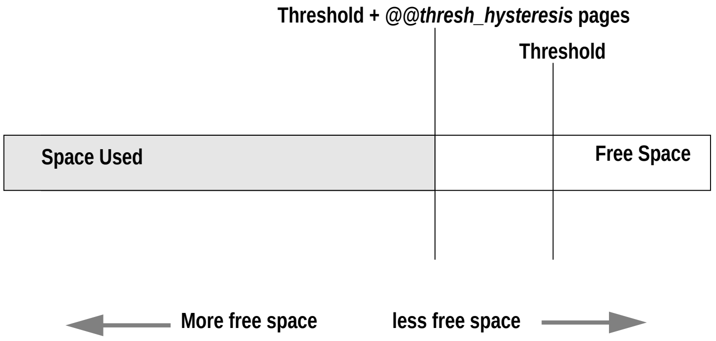


Figure 21-3: Free space must rise by @@thresh_hysteresis to reactivate threshold

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**, **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 suspend 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 **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 unsuspend function with extreme caution. If you wake a suspended process with lct_admin and do not immediately kill it, the process may completely fill the log before the dump completes.

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 4-13 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 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 **sp_helpthreshold** shows that there is one threshold on this segment set at 200 pages. The 0 in the "last chance" column indicates that this is not a last-chance threshold:

```
     sp_helpthreshold "default"
segment name   free pages   last chance?   threshold procedure
------------   ----------   ------------   -------------------
default        200                     0   space_dataseg

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

Adding a Threshold

Use the system procedure **sp_addthreshold** to create new thresholds. Its syntax is:

sp_addthreshold ***dbname, segname, free_space, proc_name***

The *dbname* 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 when 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 when 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 dbname, segname, free_space
   [, new_proc_name [, new_free_space
   [, new_segname]]]
```

where *dbname* is the name of the current database, and *segname* and *free_space* identify the threshold that 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_space* 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 dbame, segname, free_space
```

The *dbname* 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 blocking user transactions).

The additional threshold should dump the transaction log often enough 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 percent of log capacity and adjusts this threshold based on space usage at your site.

Adding a Log Threshold at 50 Percent of Available Space

Use the following procedure to add a log threshold at 50 percent 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 percent 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 "Creating Threshold Procedures" on page 21-12.

Testing and Adjusting the New Threshold

Use the **dump transaction** command to make sure your transaction log is less than 50 percent 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 percent 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 considerable space was left in the log when the dump completed, you may not need to dump the transaction log so soon:

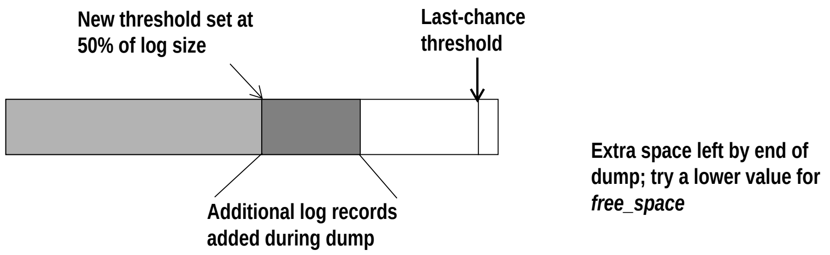

New threshold set at 50% of log size
Last-chance threshold
Extra space left by end of dump; try a lower value for *free_space*
Additional log records added during dump

Figure 21-4: Transaction log with additional threshold at 50 percent

Try waiting until only 25 percent of log space remains:

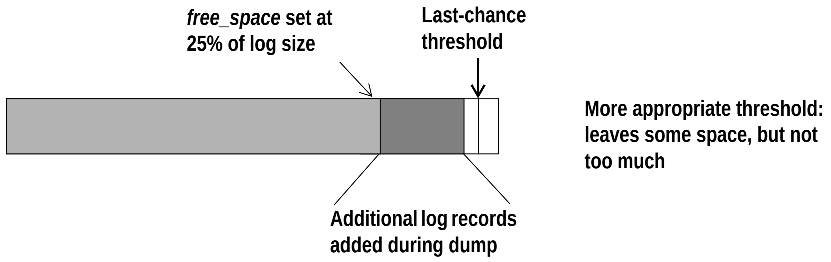

free_space set at 25% of log size
Last-chance threshold
More appropriate threshold: leaves some space, but not too much
Additional log records added during dump

Figure 21-5: Moving threshold leaves less free space after dump

Use **sp_modifythreshold** to adjust the *free_space* value to 25 percent 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:

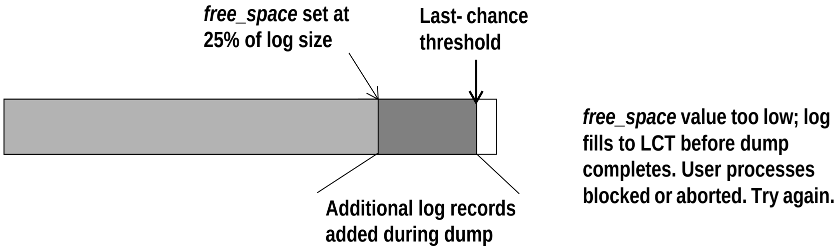

free_space set at 25% of log size
Last- chance threshold
free_space value too low; log fills to LCT before dump completes. User processes blocked or aborted. Try again.
Additional log records added during dump

Figure 21-6: Additional log threshold does not begin dump early enough

25 percent free space is not enough. Try initiating the dump transaction when the log has 37.5 percent free space:

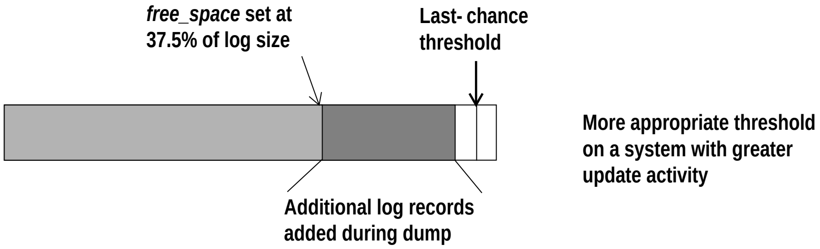

free_space set at 37.5% of log size
Last- chance threshold
More appropriate threshold on a system with greater update activity
Additional log records added during dump

Figure 21-7: Moving threshold leaves enough free space to complete dump

Use **sp_modifythreshold** to change the *free_space* value to 37.5 percent 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:

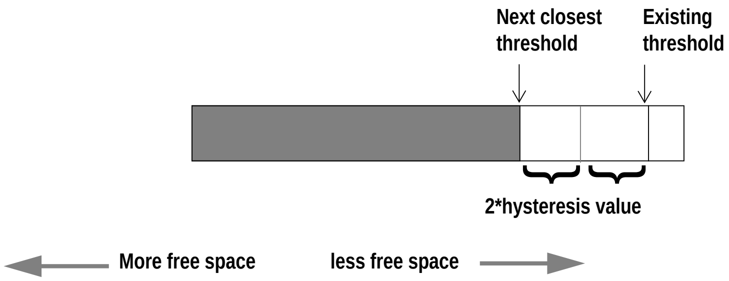


Figure 21-8: Determining where to place a threshold

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 want to dump to disk and then copy the resulting files to offline 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 the 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 threshold procedure below 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 threshold 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 easier to create a single threshold procedure that is executed by all 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.

Glossary

allocation unit

A logical unit of 1/2 megabyte. The **disk init** command initializes a new database device for SQL Server and divides it into 1/2 megabyte pieces called allocation units.

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

The end-of-batch signal that sends the batch to SQL Server for processing.

context-sensitive protection

Protection that provides certain permissions or privileges depending on the identity of the user. This type of protection can be provided using views and the user_id built-in function.

conversion

See **character set conversion**.

data definition

The process of setting up databases and creating database objects such as tables, indexes, rules, defaults, 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

One of the components of a database: table, view, index, procedure, trigger, column, default, or rule.

Database Owner

The user who creates a database. A Database Owner has control over all the database objects in that database. The login name for the Database Owner is "dbo".

datatype

Specifies what kind of information each column will hold, and how the data will be stored. Datatypes include *char*, *int*, *money*, and so on. Users can construct their own datatypes based on the SQL Server system datatypes.

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 connects with 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 2K pages.

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

An computation, column data, a built-in function, or a subquery that returns values.

extent

Whenever a table or index requires space, SQL Server allocates a block of 8 2K pages, called an extent, to the object.

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

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 0–9 and the uppercase and lowercase letters A–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

A string of characters used to identify a database object, such as a table name or column name.

initialization

See **disk initialization**.

int

A signed 32 bit integer value.

intent lock

Indicates the intention to acquire a share 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 SQL 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 a **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.

non-repeatable read

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, so that you can perform such tasks as creating files, running programs, and printing documents.

optimizer

SQL Server code that analyzes queries and database objects and selects the appropriate query plan. The SQL Server optimizer is a cost-based optimizer: it estimates the cost of each permutation of table accesses in terms of CPU cost and I/O cost.

parameter

An argument to a stored procedure.

permission

The authority to perform certain actions on certain database objects or to run certain commands.

phantoms

Occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, **update**, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.

precision

The maximum number of decimal digits that can be stored by *numeric* and *decimal* datatypes. The precision includes **all** digits, both to the right and to the left of the decimal point.

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

The maximum number of digits that can be stored to the right of the decimal point by a *numeric* or *decimal* datatype. The scale must be less than or equal to the **precision**.

schema

Consists of the collection of objects associated with a particular schema name and schema authorization identifier. The objects are tables, views, domains, constraints, assertions, privileges and so on. A schema is created by a **create schema** statement.

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 authorized to handle SQL Server system administration, including creating user accounts, assigning permissions, and creating new databases.

system databases

The databases on a newly installed SQL Server: *master*, which controls user databases and the operation of the SQL Server; *tempdb*, used for temporary tables; *model*, used as a template to create new user databases; and *sybsystemprocs*, which stores the system procedures.

system function

A function that returns special information from the database, particularly from the system tables.

system procedures

Stored procedures that SQL Server supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from the system tables, or mechanisms for accomplishing database administration and other tasks that involve updating system tables.

system table

One of the data dictionary tables. The system tables keep track of information about the SQL Server as a whole and about each user database. The Master Database contains some system tables that are not in user databases.

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.

- Numerics

  Indexes entries that begin numerically.

- Subjects

  Indexes subjects alphabetically.

Page numbers in **bold** are primary references.