Microsoft

SQL Server 2000 evaluation software inside!

Microsoft® SQL SERVER™ 2000 HIGH AVAILABILITY

Allan Hirt with Cathan Cook, Kimberly L. Tripp, and Frank McBath

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2004 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Microsoft SQL Server 2000 High Availability / Allan Hirt ... [et al.].
p. cm.
Includes index.
ISBN 0-7356-1920-4
1. SQL server. 2. Client/server computing. I. Hirt, Allan.

QA76.9.C55M53215 2003
005.75'85--dc21 2003051241

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 8 7 6 5 4 3

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com*.

BizTalk, Microsoft, Microsoft Press, Outlook, Visio, Visual SourceSafe, Windows, Windows NT, Windows Server, and Windows Server System are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: Kathy Harding
Project Editor: Maureen Williams Zimmerman

Body Part No. X09-39086

Table of Contents

9

Database Environment Basics for Recovery

Whether you implement failover clustering, log shipping, replication, or a combination of these, they cannot supplant a solid backup and recovery plan. Things can—and will—go wrong. Even a well-planned highly available system is subject to user error, administrative error, procedural failure, or a catastrophic hardware failure. Creating, testing, and maintaining a database environment in which little to no data is lost and downtime is entirely avoided in a disaster is no trivial task. Because backup and restore are important, required parts of any disaster recovery plan, your backup and restore strategy should minimize both data loss and downtime. This chapter gives you the basic understanding to be able to proceed to Chapter 10, "Implementing Backup and Restore," where you learn how to implement a backup and restore plan in your environment.

Fundamentals

No matter what size the database or the availability requirements, restore is always an option. In some cases, such as in the event of accidental data modifications or deletions, it is the only option that lets you restore the database to the state it was in before the modification. But what does it mean to have a backup and restore strategy focused on high availability?

To be focused on high availability you must be focused on whether or not the system is accessible and, if it is not, on how long it will be down. A successful disaster recovery plan lets you recover your database within your company's defined acceptable amount of downtime and data loss. If downtime must be kept to an absolute minimum, the key requirement for your backup strategy is

recovery speed. How do you make your recovery fast? Are there database and server settings that can impact the recovery of your database? What options must be determined as part of your strategy? There are many possible backup and restore strategies, each of which offers different levels of trade-offs between hardware costs, administrative complexity, potential work loss exposure (data loss in the event of a failure), performance during the backup, performance of batch operations or maintenance operations, as well as day-to-day performance of user operations around the clock. Finally, the options that you choose to employ could also have effects on the transaction log in terms of active size, backup size, and whether log backups are negatively affected during other operations.

For example, did you know that log backups are paused while a full backup is running? Do you know the impact of pausing log backups on other features that depend on frequent log backups, such as log shipping? Do you know what else could go wrong if the log backup does not occur? The decisions you make here are truly critical to the overall success of your recovery plan. But where do you start?

Technology Last

First, you must know the barriers for which backup and restore will be the solution. This helps you determine where you are at risk. Second, you must know your environment. Knowing your data; the database structures; how the data is being used; changes made to database settings throughout the day, week, or month; and the acceptable amount of downtime and data loss (which could vary at different times of day) is important. You must be familiar with user, administrative, and batch processes so that you are aware of all that could fail and what was happening at the time of the failure. This information will help you estimate how much time is acceptable for repair and recovery, which in turn helps to dictate your hardware choices. Third, you must have a recovery-oriented plan that fully aligns with the process of database recovery and the restoration phases.

Using these key facts, you can decide among the backup types available and come up with the best strategy for your environment. Only after you fully understand each one of these factors should you determine which backup strategy is best. Unfortunately, administrators commonly define backup strategies by learning only the backup technologies available without considering recovery. This is exactly the wrong approach; instead, you should consider technology last and recovery first. If you do not have a recovery-oriented plan, you will more than likely suffer data loss and significant downtime.

Understanding Your Backup and Restore Barriers

Whether you are recovering from accidental data deletion, hardware failure, natural disaster, or other unplanned incident, you will want your recovery to be well thought out. There are really two categories of barriers that are likely to be overcome with your backup and restore strategy: hardware failure and application or user error.

Hardware Failure

You should not use backups to recover from hardware failure as a common practice, as it is likely your system already has hardware redundancy in place. Whether you are trying to set up a highly available server or just a production database server, you should always start by using some form of disk redundancy, such as RAID. In Chapter 4, "Disk Configuration for High Availability," you looked at many disk considerations for the foundation of your database, and it is likely you have chosen some form of mirroring, striped mirrors, or striping with parity. However, what if you lose an entire RAID array? What if a single disk is lost in a RAID 5 set and the administrator replaces the wrong disk during the hot swap to replace the failed disk? In the case of hardware failure, you might choose to minimize downtime by bringing a secondary or standby server online; however, you will need to recover the failed primary. Often, this is done with backups.

User Error

You might think that your job would be great if there were no users or even other DBAs or system administrators. Quite frankly, no human intervention of any sort would be preferable for most! If you could create a database and then *never* use it, it would be much easier to manage. Nevertheless, if users can modify data, inevitably someone at some time will modify something incorrectly. Similarly, if system administrators have direct access to the production server, they have the ability to directly change your production data. In Chapter 14, "Administrative Tasks to Increase Availability," you will look at the administrative processes that should be in place to maintain and secure a highly available system, but even with extensive preparations, accidents will happen. In fact, accidental damage is the most difficult from which to recover and it can spread much further than just an incorrectly dropped table. Application, user, and process error could occur almost anywhere. Examples include the following scenarios:

- Administrators or database owners (DBOs) dropping a table incorrectly because they are connected to their production and development databases all day long—within the same tools.

- Users accidentally modifying the wrong data because they have direct base table permission to INSERT, UPDATE, and DELETE, and although they normally remember a WHERE clause, they forget to highlight it when executing their query.
- Batch processes accidentally dropping the wrong database because the script performs a drop and re-create of the database. It is the first time the script is being run on that server, where a different database—named the same as the other database but supporting different functionality—already exists.
- Batch processes accidentally creating objects or making changes to the wrong database because the initial database creation fails due to a path error or a not enough disk space error. With little or no error handling in the script, it continues to run incorrectly in the wrong database. All of the script's objects end up in the connected user's default database, which in this case is set to the master database.

All of these scenarios are possible, and these few examples are really only the tip of the iceberg. More important than the original failure is the recovery process that follows. Incorrect data modifications are the most difficult to recover from because the longer any problem is left unmanaged, the more likely you are to lose data. Additionally, the longer you wait, the more difficult it is to recover the data from the still potentially changing database.

How quickly do your users come running down the hall or pick up the phone to tell you about their accidental data deletion? Does the DBA immediately refer to the disaster recovery plan when he or she makes a mistake, or does he or she try to troubleshoot the problem, possibly compounding it? What is the best plan of recovery and, more important, how can you prevent some of these mistakes from happening in the first place?

Minimizing Human Error

To make a system both secure and highly available, you need to have administrative change control as well as maintenance processes in place to minimize direct access to production databases. There is a common question asked by DBAs: Is there a way that SQL Server system administrators can be prevented from dropping tables? This sometimes garners the answer, "Get new system administrators." All kidding aside, this can and does happen. This problem is so common that many system administrators have learned quite a few tricks, and the next several paragraphs include a few ideas—not specific to backup and restore—picked up from them along the way.

To prevent tables from being incorrectly dropped, consider schema-bound views using declarative referential integrity (DRI), which makes inadvertently dropping a table more difficult. However, DRI prevents a dropped table only when the table is being referenced with a foreign key constraint. A sales table, for example, often references other tables, but other tables do not always reference it. So how can you prevent an accidental table drop? Consider using a schema-bound view. If a view is created with SCHEMABINDING, then the table's structure cannot be altered (for all columns listed in the view), and the table cannot be dropped (unless the view is dropped first).

> **More Info** For details on how to create schema-bound views in Microsoft SQL Server, see Instant DocID#22073 on the *SQL Server Magazine* Web site at *http://www.sqlmag.com*. This article does not require a subscription.

To prevent data from being incorrectly modified, consider eliminating all direct access to the base tables. Often having applications designed to manipulate the data are best; however, users might then require direct ad hoc access to your data. Is it really necessary? This requirement usually indicates that the users are not getting the information they need and the developers gave up. Regardless, creating boundaries within the ad hoc environment is better than complete chaos that is marked by bad queries, poor performance, and unhappy users. Instead of granting direct access (SELECT, INSERT, UPDATE, and DELETE) to the base tables, create views, stored procedures, and functions to handle the data access. Using these objects, you can add error handling, trap unwanted change, manage data redundancy, and generally prevent accidental modifications where a WHERE clause has been left off inadvertently.

If this seems like a lot of work, you might be surprised at the secondary benefits. Typically when users write ad hoc queries, performance suffers because of mistakes in writing Transact-SQL. Users who do not write a lot of Transact-SQL code are prone to writing poorly performing code. You can optimize objects through the development and quality assurance testing of the views, procedures, and functions, providing a better outcome for everyone.

Finally, to prevent mistakes in batch processing, consider error handling and a scheduled code review with all key personnel. The code review allows other experienced DBAs to determine if anything could interfere with what they are responsible for. Additionally, another set of eyes to review the batch could prevent something that might otherwise be a problem with running the batch in the existing

environment. This need not be a line-by-line code review (although it could be), but it should at least explain the general principles behind the script's execution. Give special attention to all components of the script that drop or modify already existing data, objects, or databases. If the script includes proper error handling, the time for the code review could be reduced.

As a trick in batch processing, consider using RAISERROR. There is a special value for the state parameter of RAISERROR that might help by forcing the termination of a complex script and preventing further execution when the script is processing incorrectly. Raising an error with a state of 127 causes the script to stop processing, and this can be especially helpful when the script might end up processing in the wrong database. However, setting the state value does not always appear to terminate the session. Applications such as SQL Server Query Analyzer might automatically reconnect when a connection is broken. To fully realize the benefit of the state option, you need to use a tool such as Osql.exe, which does not reconnect automatically after the connection is terminated.

> **More Info** For details on some of the benefits of using RAISERROR in your Transact-SQL statements, see Instant DocID#22980 on the Web at *http://www.sqlmag.com*. This article does not require a subscription.

However, be aware that nothing is guaranteed. Even if you prevent many errors using these techniques, the database probably will still need to be recovered after some form of human or application error.

Symptoms and Recovery

Recovering a database after hardware failures or incorrect data modifications can be quite complex, as there are numerous elements that can fail. Even more numerous are the options for recovery. The failure might be isolated to one disk, one RAID array, one table, a group of tables, or only part of the data. Remember, to create a strategy for high availability you want to recover as fast as possible. If the damage is isolated, can your restore and recovery be isolated? Possibly, if you plan for it.

Backup

Before creating your backup strategy, there are a few key facts to know about how SQL Server works with regard to backup and recovery. First and foremost,

there is no need for a traditional backup window. Backups can occur concurrently with other operations and while users are online actively changing the data. Backups have very little impact on the existing workload, as they do not rely on reading the active data (this is discussed in detail later). Additionally, backups run as fast as the hardware allows and they are self-tuning.

With this in mind, you might wonder why you cannot just perform backups constantly, and when one completes, begin another. There are many reasons why this might or might not be a good idea, and this is what this chapter is about: knowing the basics before you implement your backup and restore plan. More important, are you familiar with the technology? Are you aware that some backups conflict with some administrative operations? For example, when performing a full database backup, you can neither change the database's file and filegroup structure (either manually or through autogrow options) nor back up the transaction log. These limitations might prove significant as you review possible backup strategies.

Understanding Database Structures

Every database has a data portion and a log portion. When you define the database you must create a log—whether you want one or not—and you cannot turn logging off. In fact, SQL Server creates a transaction log file for you if one is not specified. When the size of the log is not specified, the default size varies between the syntax and the Enterprise Manager dialogs so it is important to explicitly state the size (the Create Database dialog uses a default of 1 MB for the transaction log and the CREATE DATABASE syntax defaults to 25 percent of the total size of the data portion; neither is usually appropriate). However, sizing the transaction log is not easy, as there are numerous factors on which the log size is based. One of the most important is related to backup.

> **Warning** Do not use file system compression with SQL Server data files, as it is not supported. When SQL Server writes to the transaction log, it needs to guarantee sector-aligned writes. When using compressed volumes, SQL Server loses the ability to guarantee exact placement of data within a sector (a sector is 512 bytes, and SQL Server writes in 8 KB blocks). Putting SQL Server data files on compressed volumes has resulted in lost data. For more information, review Microsoft Knowledge Base article 231347, "INF: SQL Server Databases Not Supported on Compressed Volumes," found at *http://support.microsoft.com*.

Understanding the Write-Ahead Log

When modifications are written to the database, SQL Server goes through a series of steps to ensure consistency and recovery. To ensure consistency, SQL Server takes the necessary locks; to ensure recovery, SQL Server writes information to the transaction log portion of the database. A simplified version of the process is described in the following steps and text. Although this version is simplified, it will give you a good understanding of how the log is defined as a write-ahead log and why it is important for both SQL Server and manual recovery.

1. A user submits a single data modification statement—for example, an update. This update will affect five rows out of the one million rows within that table. This is considered an implicit transaction as all five rows need to be modified or the transaction will not be complete.
2. SQL Server begins the modification by taking update locks on all of the rows required (there are other locks at the page level, table level, and database level; however, for this example, they are not significant). An update lock is an interesting lock that represents someone who has the intent to modify but has not yet modified the row. SQL Server can proceed to perform the changes only after all update locks on all rows involved take place. Although the rows have only the update lock (as SQL Server is repeating Step 3 for each row), rows with an update lock are accessible to readers, allowing better concurrency.
3. Modifying the data actually occurs in a number of steps. For each row, SQL Server follows this process:
    a. It obtains the exclusive lock, which is sometimes referred to as X. To perform the actual data modification, SQL Server must guarantee that no one else can see this data (that is, no one can access this "dirty" and uncommitted data). To do so, a stricter lock is required (an exclusive lock). The exclusive lock specifies that only *this* transaction can access this data exclusively until the transaction no longer needs it (once the transaction is committed).
    b. SQL Server modifies the row (performs the modification as defined by the modification statement).
    c. It logs the modification to the transaction log (which for this modification might solely be in memory at this point). However, once all of the rows have been modified, the next step is to commit this transaction and make it recoverable.

4. Once all of the modifications have been performed, the transaction is ready to commit. The process of committing the changes also occurs in a number of steps:
   a. SQL Server writes the changes to the transaction log on disk (this might have already happened, but if any of the log pages on which this transaction resides are still in memory, they are written at this point).
   b. It releases the locks.
   c. It notifies the client that the modifications are complete. The user receives the "5 Rows Affected" message.

This information is interesting, because it shows that there is a time when information about a transaction is only in memory (Step 3) and another time when it is both in memory and its log changes are on disk (Step 4). When does the data make it to the data portion of the database? A separate process synchronizes changes from memory to their appropriate locations on disk. This process, called a *checkpoint*, really exists to synchronize *all* dirty pages with their appropriate location on disk, regardless of the state of the transaction (this is one of the reasons a log page might have already been written to disk before Step 4). Log pages are always written ahead of data pages, and in many cases it might be minutes ahead. A checkpoint is a batch operation (not to be confused with a bulk operation) that allows modified pages to be written to disk quickly in batches rather than as they occur. How long does it wait? That is dependent on SQL Server, and even though you can change this, it is not recommended.

> **More Info** If you are interested in learning more about this configuration option, check out the Recovery Interval setting in SQL Server Books Online and in the Microsoft Knowledge Base.

The checkpoint process is batched to minimize thrashing to disk, as some data pages might change due to numerous transactions over a short period of time. Instead of writing those pages to disk as they occur, the checkpoint batches them, minimizing the writes to disk. However, to guarantee a user that his or her transaction is in the database, even if there is a power failure (remember, the user has received the "5 Rows Affected" message), SQL Server writes

transaction log information at the commit of a transaction. This allows the transaction log activity to be predominantly sequential writes and the data portion (except at checkpoint) to be more random reads (yes, some users read large sequential amounts of data, but with numerous users it is more random).

In summary, this is why the transaction log is called a write-ahead log. The log information is written to disk on the commit of a transaction ahead of the data. The data is written later during a checkpoint. To improve performance, you can always place the transaction log portion of your database on a dedicated disk. This can improve performance, and it is also important for recovery because it allows you to set different options for the drives on which these files are located. So how does this apply to backup and restore in terms of recovery?

For databases to recover from incidents like power failure, transactional information must be available. The transaction log provides this set of instructions, which can be used in recovery. The instructions contain what has changed within your database, and this always gets you from one version of the data to another, not unlike how driving directions get you from one location to another. Have you ever wanted to tell someone how to get somewhere? When you are talking to them you always start with a point of reference. For example, starting at X, you take Y to Z, and so on. The same is true for the transaction log. All transaction log entries act as instructions based on how the data looked at the time. To use those instructions, the system must have the data in the same state as it was in at the time, or the instructions will not make sense.

There are really two purposes for the transaction log. Automatic Recovery is SQL Server's primary use for the transaction log. Automatic Recovery occurs each time SQL Server is started to ensure transactional consistency. If transactions were being processed when the server was stopped, SQL Server can recover those changes by accessing the log when the server restarts. This ensures that only committed transactions are within the database and uncommitted transactions are rolled back. SQL Server recovers system databases and then user databases at startup. Automatic Recovery first reads the log and goes through a phase called redo (roll forward). During this phase, the transaction log is read to find all of the changes and perform them, loading the information into the cache. Once redo is finished, SQL Server performs the undo phase. During this phase, SQL Server rolls back any changes that do not have a corresponding commit. If the server was shut down properly this should process very quickly; however, if the server suffered a power failure or improper shutdown, Automatic Recovery might take significantly longer.

During Automatic Recovery, SQL Server needs to see the instructions about the changes that are kept in the transaction log if they have not yet made it into the data portion of the database on disk. Remember, at Step 4 (above), the transaction is partially on disk and partially in memory. Specifically, the instructions

about the change are in the transaction log on disk and the result of the data changes is in the data portion of the database in memory. Once the transaction log has been read and all steps have been processed through redo and undo, the last step of Automatic Recovery is performing a checkpoint. This synchronizes memory to disk to ensure that all changes processed by Automatic Recovery are on disk and would not need to be processed again if the server were to suffer another power failure. Once recovery completes, SQL Server makes the database available.

Once the checkpoint process synchronizes that information to disk, SQL Server no longer needs the information for Automatic Recovery. In fact, SQL Server even allows you to set an option to clear the information if you choose to. However, this is where the secondary benefit of having the transaction log can be seen: manual recovery. You might need to perform manual recovery if the database is damaged.

> **Important** Although secondary to SQL Server, the advantage of being able to do a manual recovery is the most important reason to keep the transaction log and not let SQL Server clear it.

This is another reason the transaction log should be created on a physical disk that is separate from the data portion of the database. By keeping these instructions in the transaction log, you can set up a process to capture these changes later, potentially even when the data portion of the database has been damaged. To capture these changes, you perform transaction log backups. In the event of a failure that renders the database inaccessible or corrupts the data in the database (due to application or human error), having the transaction log backed up gives you something external to the corrupted database from which you can recover, potentially even up to the time (or just prior to the time) at which the database became corrupt. In fact, you can perform backups of the transaction log at periodic intervals. Each one will act as a set of directions to get you from one point to another. The more frequently you capture these instructions, the closer and closer you can get without directly accessing the original data.

In total, a complete sequence of log backups can get you from various starting points up to the time of a failure, especially if you have the final set of instructions. Once you have captured these instructions, SQL Server removes the inactive instructions (instructions from transactions still being processed cannot be removed) from the transaction log portion of the database. This helps

to maintain the overall size of the transaction log. Second, by frequently capturing these instructions, you create smaller backups. In turn, these smaller transaction log backups can be performed with little impact on users. Because the transaction log is critical to recovery, you should make an effort to make it as efficient as possible.

Optimizing the Performance of the Transaction Log

The log portion of the database should have exactly one file (maximum number of total files—both data and log—is 32,767); there is rarely a need for more than one transaction log because you will not see any performance benefits. More than one transaction log could be useful only if your log needs to span multiple volumes. If you have more than one file based on capacity alone, you should consider increasing the frequency of transaction log backups so that a buildup of instructions does not occur. Also, consider using hardware RAID to handle the increased need for capacity over having multiple log files. Increasing the frequency of log backups not only minimizes the need for a large transaction log, it also minimizes your potential data loss exposure.

Because the transaction log is critical in most recovery scenarios, it is also important to make sure that the drive on which the transaction log resides is also on some form of RAID. RAID 1 mirroring is acceptable if the transaction log is not overly active. For extremely active transaction logs where disk activity queues or where performance is not optimal, consider using a combination of mirroring and striping (preferably striped mirrors) for the transaction log. This might mean giving a significant amount of disk space to a relatively small amount of information, but therein lies the trade-off of disk space versus performance (and possibly availability).

Even if the transaction log is only one physical file, as recommended, SQL Server maintains that file internally as multiple virtual log files (VLFs). At the creation of a database, the transaction log is divided into multiple VLFs. SQL Server determines the size of the VLFs, which is not generally interesting to the typical administrator. However, you should be concerned with how many VLFs get created because you do not want to create fragmentation and noncontiguous log access. If the transaction log size was properly estimated during capacity planning and the database was set up with the appropriate size when it was created, the number of VLFs will be optimized for the size of the file. If the file was properly created at the correct size it will have very few VLFs.

For example, a 1-GB transaction log will have only eight VLFs. Again, having an optimal number of VLFs is based on the file being initially created at (and not autogrowing to) 1 GB. If the file is added at only 100 MB and grows to 1 GB, you end up with significantly more VLFs; in fact, at least one VLF for each autogrowth. With a transaction log that grows automatically by 10 percent,

starting the transaction log at 100 MB and growing it to 1 GB would create roughly 25 VLFs instead of 8. (In some cases you might see hundreds of VLFs when autogrow is growing by a smaller amount, more frequently.) Having more than the necessary number of VLFs adds overhead both in terms of backup performance and transaction log performance (logging).

To minimize the number of VLFs, you need to define the transaction log size appropriately (or at least reasonably) at creation. For the databases where files are set to autogrow, you can also minimize the number of VLFs by setting the autogrowth size to a reasonable fixed number of megabytes (the default is 10 percent, but this requires calculation, as it might not be enough). Additionally, transactions processing at the time of autogrowth are paused (or blocked) and might time-out based on your client settings. Instead, actively monitoring your transaction logs, performing frequent transaction log backups, and minimizing long-running transactions gives you a more appropriate number of VLFs and better performance.

To see the number of VLFs your database's log file has, execute DBCC LOGINFO.

> **Caution** DBCC LOGINFO is an undocumented READ ONLY command that displays information about the transaction log. There is no guarantee that it will work as defined here in future releases.

For the purpose of this discussion, all you are interested in is the number of rows—this indicates the number of VLFs. If you have more than 16 VLFs, you should consider trying to consolidate them. The best way to do this is to clear the transaction log with a regular transaction log backup (if being performed), then shrink the transaction log with a DBCC command and then finally, manually set the size to the more appropriate size through one execution of ALTER DATABASE (instead of numerous autogrowths).

In Microsoft SQL Server 2000, shrinking the transaction log *should* occur easily by performing a regular transaction log backup and then using DBCC SHRINKFILE (*logfilename*, TRUNCATEONLY) to shrink the transaction log to the smallest possible size. Immediately following the SHRINKFILE, execute ALTER DATABASE, and increase the transaction log's size as appropriate. Make sure to set the maximum size to a finite value instead of allowing it to be unlimited. Additionally, you should make sure to monitor the log size and create both jobs and alerts to regularly manage the transaction log.

Understanding Continuity of the Transaction Log

Even though recommended recovery strategies are discussed later, along with how the different backups work, it is important to make sure that you completely understand the importance of maintaining continuity of the transaction log. The database's transaction log contains instructions and a transaction log backup allows those changes to be recorded in another location (a backup device). When a transaction log is backed up, SQL Server can clear much of what was backed up because it no longer needs the information for Automatic Recovery. Due to this clearing (or truncation), transaction log backups are usually instructions only since the last log backup (there is an option that allows you to back up a transaction log and not clear it; however, this is not the default and it is typically used only in special circumstances). Other backups do not affect the transaction log; only transaction log backups manage the transaction log.

Take the following example in which full database backups are represented as F_1 and F_2 and 20 transaction log backups are shown as l_1 through l_{20}. Consider the group of backups, starting with each of the full database backups, as a recovery set. One recovery set begins with F_2 and the previous begins with F_1.

F_1 l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8 l_9 l_{10} l_{11} l_{12} **F_2** l_{13} l_{14} l_{15} l_{16} l_{17} l_{18} l_{19} l_{20}

This example shows a total of 22 backups. With these two recovery sets you have created multiple recovery paths. The optimal recovery path would be to recover with the last full database backup and then apply all of the remaining transaction log backups.

F_2 l_{13} l_{14} l_{15} l_{16} l_{17} l_{18} l_{19} l_{20}

What if the full database backup F_2 were bad? Do you have any other options? Yes, you do. This is the beauty of the design of both the transaction log and transaction log backups. If the full database backup at F_2 is bad, you can recover using the F_1 full database backup instead. At F_1 you can apply the entire series of transaction log backups in sequence and still recover up to transaction log l_{20}. If recovery from F_1 was desired, the restore sequence would be as follows:

F_1 l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8 l_9 l_{10} l_{11} l_{12} l_{13} l_{14} l_{15} l_{16} l_{17} l_{18} l_{19} l_{20}

In fact, even if the full database backup F1 was bad, you could go back to the previous full database backup—assuming it is still available and all of the transaction log backups are accessible—and you could still roll forward to the last transaction log backup at l_{20}.

Many interesting observations stem from this example. As transaction log backups are performed, SQL Server completes the backup by essentially clearing (truncating) what was backed up (with the exception of that which is still active). Transaction log backups have a very specific sequence to them and you must perform recovery using *all* transaction logs. Each and every transaction log must be applied in sequence; if one is not available, the recovery process cannot move forward. More important, even when other backups are performed, they do not affect the continuity of the transaction log. The full database backup performed at F_2 can be skipped as though it had not occurred, and recovery can occur seamlessly by beginning with the full database backup F_1 and then restoring all of the transaction logs from l_1 through l_{20}.

You can learn numerous lessons from this discussion. First and foremost, transaction log backups are the most critical backups to have in a recovery scenario. If a transaction log backup is damaged, the last successfully loaded transaction log will be the final transaction log backup to which you can recover. In fact, you might even consider creating multiple copies of your transaction log backups or mirroring them to multiple backup devices. That said, with certain backup hardware you might be able to back up transaction logs to multiple devices simultaneously. In Figure 9-1, a single transaction log backup is written to four tapes.

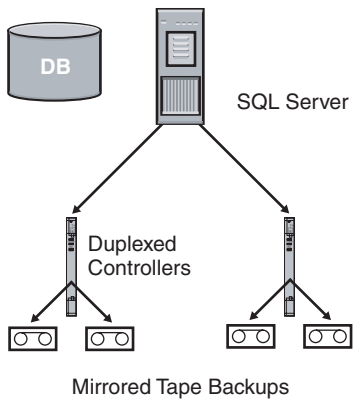

Figure 9-1 Duplexed and mirrored transaction logs.

It is also important that you keep more than one set of backups on hand. Before you discard a recovery set's starting point (that is, before you discard a full database backup), you should test the recovery set you are keeping. To ensure comprehensive recovery of a database during a restore you must have a *complete* sequence of all transaction logs up to the time of the failure or the

point in time to which you want to recover. Having all of these logs gives you continuity. Anything that breaks the continuity of the log causes that to be the last log you can apply.

Other operations could affect the continuity of the transaction log, but they are not recommended when you want to recover from the transaction log. In fact, in a production transaction-processing database, the only operation that should ever truncate a transaction log is a transaction log backup. When this is true and you maintain transaction log continuity, you have added redundancy to your backup strategy.

What If the Transaction Log Fills?

Nothing is certain, and even with precautions the transaction log could still fill. This might not seem like it could be problematic but it is very important to realize that a full transaction log translates to downtime. When the transaction log is full, the database stops all modifications, and any transactions pending at the time the log fills are rolled back. Additionally, new modifications are not allowed. By most definitions this is downtime. The database is available, but only for read operations. No new modifications are allowed until space is made available. The correct—and simple—response to the full transaction log is to perform a backup of the transaction log so that SQL Server can clear the inactive portion of the transaction log (something that occurs as part of a transaction log backup).

In previous releases of SQL Server, a special command was used in this scenario, but it was mainly because of how previous versions were designed. When the transaction log filled, there was no room for SQL Server to mark (or log) the fact that the transaction log was being backed up. Because of this, transaction log backups were not allowed when the transaction log filled. The special clause WITH NO_LOG had to be added to the BACKUP LOG command to clear the log without backing it up. In Microsoft SQL Server 7.0, this was fixed and BACKUP LOG now works for most transaction logs, even when they are full.

For those of you experienced with previous releases, this might come as quite a shock, but this reaction—to flush the transaction log when it fills—is not only the *wrong* response, it should be avoided! If you flush the log, you are doing something worse than just throwing it away: you are breaking the continuity of the transaction log. In fact, in SQL Server 7.0 or SQL Server 2000, the BACKUP LOG with the NO_LOG option is really no longer necessary; it has become synonymous with BACKUP LOG WITH TRUNCATE_ONLY. You should never need either of these commands in a properly maintained database.

However, because so many people have this improper response to a transaction log filling, which is usually automated with a SQL Server Agent job,

a trace flag was added. Trace flag 3231 makes *both* of these backup commands benign. You can turn on a trace flag in multiple ways. First, you can set it as a startup parameter. The easiest way to do this is through the Enterprise Manager. Right-click your server and choose Properties. In the General Tab, click Startup Parameters. In the Startup Parameters tab, enter **-T3231** in the Parameter text box and then click Add. This will be set the next time you restart SQL Server. If you want to turn off the trace flag permanently you can remove the startup parameter. If you want to turn off the trace flag temporarily you can just execute DBCC TRACEOFF(3231). This statement turns the trace flag off until you turn it back on with DBCC TRACEON(3231) or until you restart your server (if it still remains a startup parameter).

Trace flag 3231 protects the continuity of the transaction log from common and inappropriate reactions to a full transaction log. In fact, with this trace flag turned on neither of the following commands

```
BACKUP LOG dbname WITH NO_LOG
```

```
BACKUP LOG dbname WITH TRUNCATE_ONLY
```

does anything in databases where the recovery model is set to Full or Bulk-Logged. In fact, these commands are so unnecessary that the trace flag makes them behave as if the BACKUP LOG WITH … commands execute successfully (so that automated batches do not fail), but both commands are turned into a no operation (NO-OP). With this trace flag turned on, you can ensure that the continuity is never broken by an improperly executed backup log command, minimizing the potential for human error. However, the transaction log will still be full and you still have to take the appropriate actions to resolve this.

> **Caution** Even if the continuity of the transaction log is broken, there are cases where subsequent log backups do not generate an error and are allowed. Even if a warning message is produced, it is likely to go unnoticed if transaction log backups occur through scheduled operations that continue automatically.

Unfortunately, there are rare cases when backing up the log might not be possible. Even then you still have options. If a normal transaction log backup does not work or it is going to take too much time (therefore causing downtime; remember, the database is unavailable until space is available), there are

two options. The best of them would be to add space to the size of the transaction log. There are three ways to do this: allow autogrowth, manually increase the size, or add another file to the transaction log. The easiest way is to allow the transaction log to increase through autogrowth, but make sure to set a predetermined maximum size so it does not use all disk space, and always monitor it (especially with Administrative Alerts). If the transaction log still fills, then you need to reevaluate the maximum and manually increase the size while setting a new maximum size. If the maximum is unlimited and you are out of disk space, temporarily adding another file to the log to get back up and running quickly is the best choice (and can often be the fastest, especially when the size of the file being added is significantly smaller than the current size of the transaction log). If you add a file, you should remove it once the transaction log has been backed up properly. To remove this file you can use DBCC SHRINKFILE with the EMPTYFILE option; this causes SQL Server to stop using this file for transaction log extent allocations. Once the file is properly emptied, it can be removed with the ALTER DATABASE REMOVE file option.

The other option, which is never recommended if recoverability or availability is a goal, is to flush the information that is currently in the transaction log. You should always remove the inactive entries from the transaction log with a transaction log backup. If you choose to clear the log and not back it up (maybe because nothing else is working, as you have no free space and no additional disks on which you can create another file and seemingly nothing else to do), then you will have broken the continuity of the transaction log and need to create a backup that does not require transaction log backups—either a full database backup, a differential database backup, or a complete set of file or filegroup backups (a new recovery set).

It is very important that you create a backup after the continuity of the transaction log is broken. If your database becomes corrupt before you have a chance to create a new recovery set, a disaster would possibly cause a loss of data. Without a new recovery set, the backups you have can only restore up to the last successful transaction log backup performed before the continuity of the log was broken. Every transaction log backup performed after the continuity of the log was broken is useless. Having a new full database backup or differential database backup allows you to get your system back up and running, which is the most important thing. However, you have lost the ability to go back to the previous recovery set and move beyond where the continuity of the log has been broken (with the exception of differential backups) and move forward. At this point, you have lost some redundancy in your backup strategy and it is even more important that this backup is protected and tested. This is an important point when reviewing the pros and cons of various backup strategies!

> **Important** Remember, although SQL Server provides functionality to truncate (clear) the transaction log and it is very easy to do, clearing the transaction log significantly compromises your available options during recovery. If you do truncate the transaction log, immediately perform the correct backups to create a new recovery set. Simply put, you must follow this operation with a full or differential database backup, or a complete set of full or differential file or filegroup backups.

Breaking the Continuity of the Transaction Log

In well-maintained databases, meaning those that have a well-planned backup and restore strategy and effective log monitoring, and databases for which capacity planning and testing have defined an appropriately sized transaction log, operations that clear the transaction log—other than normal log backups—should never occur. In fact, it is critical to understand what operations should not be performed. The following operations break the continuity of the log:

- Clearing the log with
  - BACKUP LOG *dbname* WITH TRUNCATE_ONLY
  - BACKUP LOG *dbname* WITH NO_LOG
- Changing the recovery model to Simple
- Discarding a transaction log backup (by overwriting it or deleting the file)
- Having a transaction log backup become corrupt

All but the last operation are controllable. Corruption of a transaction backup log is hard to control, but performing transaction log backups to hard disk is usually safer than using tape, as tapes have a higher rate of error. Mirroring the backup device (either disk or tape) where the transaction log backups are written can significantly reduce the chance of corruption. Finally, there are additional backup types that can also minimize the reliance on a significant number of transaction log backups in sequence (that is, differential). In fact, there are numerous precautions you can take when managing backups of the transaction log.

Properly Managing the Transaction Log

Proper maintenance of the transaction log is, obviously, critical to keeping the database up and running. Additionally, proper management keeps the transaction log size smaller and eliminates the need for emergency operations that could compromise your recovery. Performing more frequent log backups is the best way to minimize the potential for data loss. However, there are a few key things that could make backups of the transaction log more difficult. It is very important that you understand how the transaction log works and how a transaction log backup works. Even if transaction log backups are set to occur frequently, you might not see the full benefit of performing them frequently unless the system is designed to support frequent transaction log backups. Simply put, SQL Server can only clear inactive transactions that have completed from the transaction log. To optimize the transaction log backup process it clears everything up to the first open transaction in the log. To keep the actively processing portion of the transaction log small, you should perform transaction log backups frequently and clear the inactive portion of the log. Some operations could prevent the transaction log from being cleared. For example, SQL Server 2000 (as well as SQL Server 7.0) never backs up any of the transaction log more than once even if a long-running transaction is active through multiple log backups, so the long transaction prevents log truncation and reuse, but does not affect anything else. All of this means you should avoid certain operations that create a significant amount of log activity or those that keep the active portion of the log active, such as the following:

- Avoid long-running transactions. Consider breaking the large transactions into more manageable chunks, and consider partitioning some of your larger tables to minimize the impact to the log during table management operations such as index rebuilds (if required). Instead of performing a single update statement against the entire set, break it into smaller batches. For example, instead of changing the entire year's sales, change them month by month, day by day, or hour by hour.
- Avoid spreading a transaction over multiple batches. If there is user interaction and the user does not interact because he or she is distracted, the transaction is considered active in the transaction log until it completes. When SQL Server backs up a transaction log it clears only the inactive portion of the log. If you suspect that a user has long-running and open transactions there are a few procedures you can use to monitor these situations: `sp_who2` and DBCC OPENTRAN.

Make sure that your transaction log is sized for all operations, especially those that might occur during the hours of a full database backup. Why? If the transaction log fills while a full database backup is being performed, you cannot back up, clear,

or increase the size of the transaction log until the backup completes, meaning your database is unavailable until then. This is the case unless you try to cancel the backup, and that might not be possible—nor is it generally a good idea, as you need to restart it again later. All of these concepts come together as you understand the process of how full database backups and transaction log backups work. However, there are still other factors—database settings.

Initial Database Settings and Recovery Models

When you create a database you always begin with a copy of the model database. All database settings are inherited from the model database at creation. The recovery model is probably the most critical setting with regard to backup and recovery, and it has different default values depending on the version of SQL Server you have installed. However, with respect to high availability, the only versions of SQL Server you are likely to be using are Enterprise or Developer editions (possibly Standard, but quite a few high-availability-related features require Enterprise Edition). The engine edition of these versions returns either Enterprise or Standard.

Note If you are interested in seeing the engine edition setting, you can use the SERVERPROPERTY function. Use the following query to see which engine edition you are running:

```
SELECT SERVERPROPERTY('EngineEdition')
```

There are only three possible return values for SQL Server 2000: 1 for the Personal and Desktop Engines editions, 2 for the Standard edition, and 3 for the Enterprise, Enterprise Evaluation, and Developer editions.

Note The default setting for the recovery model of the model database is Full recovery model if you are using the Enterprise (or Standard) edition. Use the following query to see the recovery model setting of the model database:

```
SELECT DATABASEPROPERTYEX('model', 'Recovery')
```

Make sure to use DATABASEPROPERTYEX and *not* DATABASEPROPERTY. DATABASEPROPERTYEX is the appropriate function in SQL Server 2000 that includes all of the *extended* properties not available in SQL Server 7.0.

The concept of a recovery model is new to SQL Server 2000 and the logging that is performed for numerous commands is not like any other release of SQL Server. More important, even though the name of one of the new recovery models (Bulk-Logged) sounds similar to a previous database option (SELECT INTO/Bulk Copy), this recovery model does not behave exactly the same in terms of logging and recovery. You must make sure that you completely understand the recovery models or you might be surprised by some of their effects on performance, the size of the active log, the size of the backed up log (as it differs from the size of the active log), and potential work loss exposure.

Understanding Recovery Model Settings by Default

If you worked with SQL Server releases prior to SQL Server 7.0, it is likely you have seen a full transaction log. In earlier releases the transaction log was not set to autogrow, nor did it clear by default. Because of the importance of transaction log continuity, the only operation you should use to clear the transaction log is a transaction log backup. However, the behavior of the transaction log when a database is first created might surprise you. When a database is created, by default, the transaction log runs in a mode that clears the transaction log after checkpoint until you begin your recovery strategy with your first full database backup or file/filegroup backup.

After the first backup is performed in SQL Server 2000, the behavior of your log is solely dependent on the setting for your database recovery model. In SQL Server 7.0, logging was controlled by database options (Trunc. Log On Chkpt. and SELECT INTO/Bulk Copy) and the statements and tools you executed (for example, SELECT INTO and *bcp*). In fact, prior to SQL Server 2000, the ability to completely and accurately know the state of the current transaction log was hampered. Even worse, it was during recovery that people would realize how these options and operations affected them (not realizing that they had an impact on recovery operations). Often it was too late for adequate recovery, and data loss occurred. Recovery models were introduced to simplify recovery planning and tie together the importance of the log with certain activities. In SQL Server 2000, the logging of all operations is dependent on the setting for recovery model.

Understanding Log Behavior on Initial Database Creation

In SQL Server 7.0 and later versions (including SQL Server 2000), the transaction log is in a pseudo truncate log on checkpoint mode until you perform a backup. The database option will not show as being set, and in SQL Server 2000, this behavior occurs regardless of your recovery model setting. After the creation of a database the log is set to clear on checkpoint because a transaction log backup (if you were able to back it up, which you are not) would be useless. Think back to the basics of a transaction log: it is a log or report, per se, of what has occurred

within your database. This report always gets you from one point to another, almost as detailed directions to a location would. However, you always have to have a starting point of reference. The same is true for a transaction log: it always gets you from one defined point to another. If a transaction log backup were allowed after creating a database, what would be the starting point in recovery? To which backup would you apply that log? Would it be the creation of a new database? Could you create a database, at any time, and ensure that it will *always* be the same as the original database? The answer is no. If there have been any changes to the model database, then the newly created database would inherit those changes, rendering the log backup useless because you have directions from a different starting point. Instead of leaving this vulnerability, SQL Server does not allow a log backup until you have created a starting point for recovery to which a log backup would make sense. For recovery there are really two options as starting points: a full database backup or a file or filegroup backup.

To fully understand what happens to the transaction log on the creation of a database, think through this simple scenario while reviewing the recommended performance monitor counters.

> **On the CD** The code for this log behavior example can be found in the script file Default_Log_Behavior.sql.

1. Create a test database. For this exercise you can name the database anything you want; for this example, the name TestDB is used.

```
CREATE DATABASE TestDB
```

2. Create a table. This is just a simple table that allows you to add rows quickly:

```
CREATE TABLE dbo.TestTable

col1       int            identity(100,10),

col2       datetime       DEFAULT current_timestamp,

col3       datetime       DEFAULT getdate(),

col4       char(30)       DEFAULT user_name(),

col5       char(30)       DEFAULT user_name(),

col6       char(80)       DEFAULT 'This is a wide column created to
                          simulate data and therefore create log space.' )
```

3. Verify that the recovery model of the new TestDB database is set to Full.

```
SELECT DATABASEPROPERTYEX('TestDB', 'Recovery')
```

4. Start the System Monitor and add the Percent Log Used counter for the instance of TestDB (see Figure 9-2). The Percent Log Used counter is under the Performance Object of SQLServer:Databases.

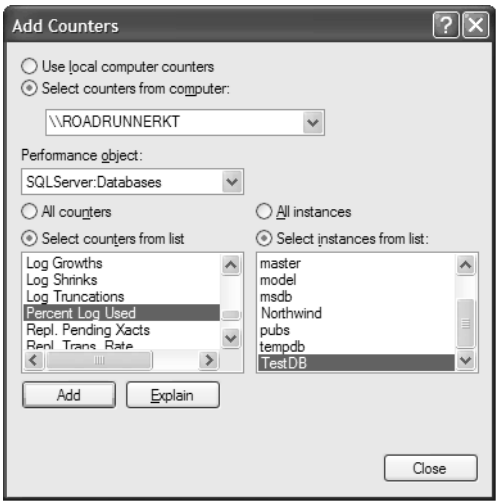

Figure 9-2 Percent Log Used counter.

5. Leave System Monitor running and create log activity with the following WHILE loop. Once you start running this infinite loop it continues to add rows to the database (autogrowing the data portion) until you manually stop the execution.

```
WHILE 1=1

INSERT dbo.TestTable DEFAULT VALUES
```

6. Return to the System Monitor and watch the log increase and then drop while the overall size of the log does not grow (see Figure 9-3). Add the Log File Size to the Performance Monitor as well. You should see that the Log File Size does not change even with all of the log activity.

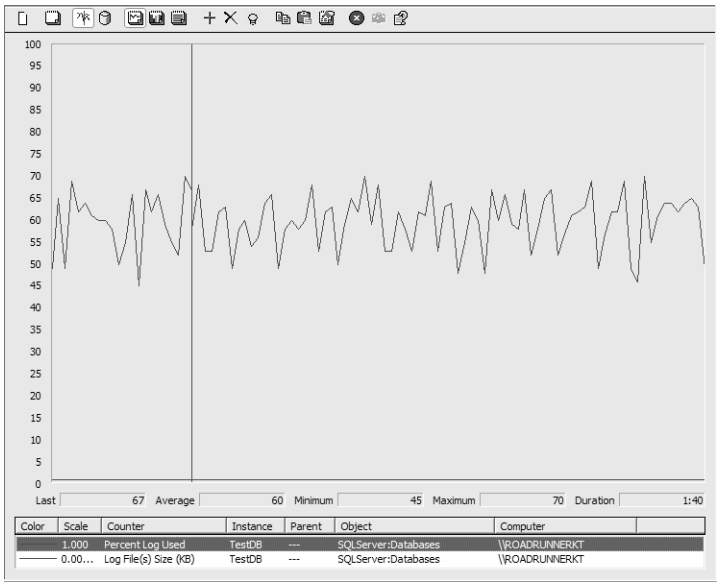

Figure 9-3 Performance of log file with autogrow.

7. Make sure you stop the WHILE loop relatively quickly; otherwise, you could fill your hard drive though database autogrowth. Remember, the database files are set to grow automatically by default.

8. Next, create a database backup using Transact-SQL. This simulates the beginning of your recovery strategy:

```
BACKUP DATABASE TestDB TO DISK = N'C:\TestDB.bak'
```

9. Run the WHILE loop again and then return to the Performance Monitor. This time you will see the Percent Log Used stay at the top of the graph, indicating that the log is almost full. You will also see that it dips, but only by roughly 10 percent, because of the automatic growth that is occurring. You will also notice that the size of the log is increasing quickly (see Figure 9-4).

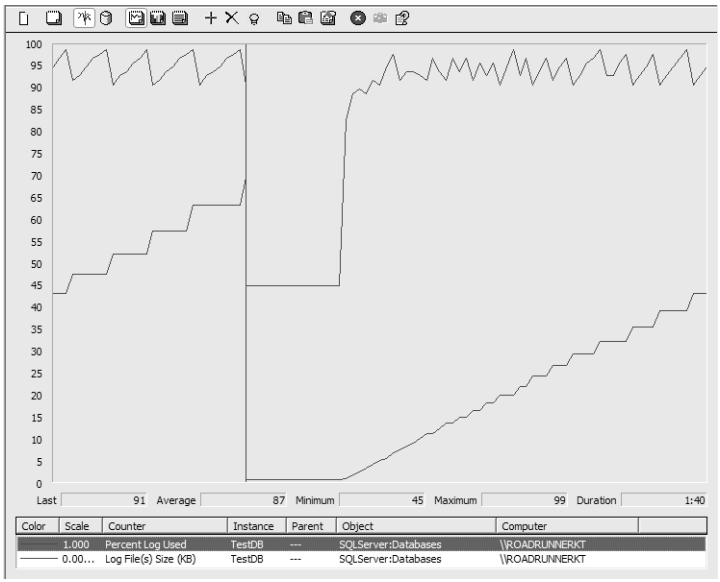

Figure 9-4 Performance of log after recovery plan is started using backup.

10. Finally, make sure that you stop the loop, delete the backup, and drop the TestDB database.

Recovery Models

There are three recovery models in SQL Server 2000: Full, Bulk-Logged, and Simple. Because there is a lot of confusion about the different recovery models, it is critical to eliminate the common misunderstandings associated with recovery models. To do this well, you should understand what the recovery models are not. Recovery models are completely new for SQL Server 2000. There are *some* similarities to former (prior to SQL Server 2000) database options, but there is no direct correlation between recovery models and previous database options. Unfortunately, this has not stopped the common comparisons between the new database recovery models and the old database options SELECT INTO/Bulk Copy and Trunc. Log On Chkpt. Table 9-1 shows the database options as they are usually compared. However, for completeness they are listed in terms of what they are not.

Table 9-1 SQL Server 2000 Recovery Models

SQL Server 2000 Recovery Model	Common Incorrect Comparisons with the Database Options SELECT INTO/Bulk Copy and Trunc. Log On Chkpt.
Full	Not the same as if neither option is set. Some operations take longer and require more log space.
Bulk-Logged	Not the same as having the select into/bulk copy option set—although there is still similar performance, the recovery has changed.
Simple	Not exactly the same as having both options set (this is the closest, however).

Truthfully, there are many resemblances between the database options and the new recovery models, but they are mostly superficial. The Full recovery model logs information in a new way. For some commands, this has never been done within SQL Server before SQL Server 2000. You might have learned this the hard way, as batch operations are likely to take more time and more log space than they did in previous releases because, for some operations, this new style of logging is more extensive.

The Bulk-Logged recovery model is also new. From a performance perspective, you can compare it to having the SELECT INTO/Bulk Copy option. However, the performance similarities exist for different reasons, as the logging has changed. In previous releases the performance gains occurred because the operations were run in a "nonlogged" state (they really were not nonlogged, as you could have still filled the transaction log; for the purposes of this discussion nonlogged means that the operation was not recoverable from information in the log). In releases before SQL Server 2000, SQL Server required these database options. Under certain circumstances the operations would run faster due to less log activity. Unfortunately, when bulk operations were performed with SELECT INTO/Bulk Copy set to true, the operation would break the continuity of the log, subsequently requiring a full database or differential database backup on completion from which you could recover. When you added the cost of the full database backup or the differential database backup to the batch operation's time, it was no longer very "fast."

In contrast, the operations run quickly in SQL Server 2000 but for different reasons: the way in which they are logged and the way in which SQL Server allows log backups on their completion. With SQL Server 2000 it is more appropriate to refer to these operations as minimally logged instead of nonlogged.

Do not be fooled, however; the performance gains of minimally logging certain operations come at a price. There is some potential for work loss if you are running the Bulk-Logged recovery model and you have performed a Bulk-Logged operation at the time of a disaster. However, if the database is accessible on completion of the bulk operation, a transaction log backup is all you need to fully recover the batch operations (this was not true before SQL Server 2000). Finally, the Simple recovery model is most like the old Trunc. Log On Chkpt. option with which the log is cleared at checkpoint, meaning that SQL Server has enough information from which committed transactions are guaranteed even after an unintended shutdown (for example, a power failure). However, because the log is cleared periodically, no log backups can occur; this eliminates manual recovery options involving the log including up-to-the-minute recovery and point-in-time recovery.

Where are the similarities between the recovery models? What should your expectations be at the time of a disaster? Using the Simple or Bulk-Logged recovery models, bulk operations should yield the same performance. However, transaction log backups are possible in the Bulk-Logged recovery model and the transaction log will not be cleared when a checkpoint occurs. You can perform transaction log backups—with the exception that the "tail" of the transaction log is not accessible after a bulk operation has been performed—to manage the transaction log size, and compared to the Full recovery model it should stay relatively small. However, the savings are only realized while the database is actively processing. The transaction log backup is significantly larger when backed up and it can only be backed up when the data portion of the database is accessible. The size of the transaction log backup should be similar in size to a transaction log backup performed when the database is in Full recovery model. However, whether or not a transaction log backup can be performed is not in question when you are running in Full recovery model. In the Full recovery model, the actual transaction log and the transaction log backup are both large. This allows you to back it up even when the database is not accessible. This is what allows the "full" range of recovery options for the Full recovery model.

Understanding the Purpose of Recovery Models

To fully understand the purpose of the recovery models, you must always speak in terms of how much and until what point data can be recovered as well as how efficiently database operations will perform. The primary focus is on what can be recovered, how it can be recovered, and what options are allowed. Before you can understand the key recovery model concepts, you must also understand the importance of the transaction log and two key concepts regarding the transaction log: how to achieve continuity (covered earlier) and how to access the "tail" of the log.

The "Tail" of the Log and Recovery Models

The transaction log is a series of instructions from the last time the transaction log was cleared (preferably when it was last backed up). If all transaction log backups are available and the current log is available then you are said to have continuity of the transaction log up-to-the-minute. What if the database is not available? If you backed up the transaction log at 3 P.M. and the database is damaged at 3:45 P.M., you have to determine up to what point data can be recovered. When a database becomes unavailable because of file corruption, disk failure, controller failure, or other situation, one of the first steps you should try to perform is a backup of the "tail" of the log. The tail of the log is the transaction log backup of all changes from the time of the last log backup until the time when the database became damaged or suspect. The tail of the log cannot be accessed if the log portion of the database is damaged. Additionally, the tail of the log cannot be accessed if the database is in Bulk-Logged recovery model and a bulk operation has occurred. If the tail of the transaction log is available, up-to-the-minute recovery is possible. Because of this, choosing the right recovery model is critical to your overall recovery strategy.

Choosing the Right Recovery Model

Generally speaking, most database recovery scenarios rely on the accessibility of the transaction log. The setting of the recovery model dictates whether or not the log is accessible and the performance impact logging has during certain operations. With SQL Server 2000, DBAs can trade performance for recovery options. Fortunately, it is completely up to the DBA and it is significantly simpler than it was in previous releases. However, in high-end databases where every transaction is critical and downtime must be at an absolute minimum, an understanding of the recovery models is required. As each recovery model is defined, keep in mind the key reasons for the addition of recovery models to SQL Server 2000:

- To better tie together the idea of the transaction log and recovery. In past releases, developers and new DBAs often learned about the transaction log after it filled. Once it filled, they also seemed to learn (very quickly) how to clear the log (that is to use the Trunc. Log On Chkpt. option), but they did not seem to learn about disaster recovery until a disaster. This happened because the option did not make it obvious that the log was needed for recovery. Now the options are database specific and their title (Recovery Model) makes it more apparent.
- To centrally and more appropriately define potential work loss exposure. DBAs should always have control over what operations are to be recoverable. In previous releases it was a complex combination of database options, user operations, and other factors that determined a database's ability to recover.

- To allow database administrators the ability to choose from among a variety of trade-offs between transaction log management, system and data recovery, and operational and batch performance for some of the more expensive operations.
- To minimize the complexity of knowing whether or not an operation can be performed. For example, SELECT INTO can be used to create a permanent table in any recovery model and no longer requires a database option. The logging of the SELECT INTO operation is determined by the database's recovery model.
- To no longer break the continuity of the log for batch processes and bulk operations, yet still allow the operations to perform optimally with only a transaction log backup required on their completion.
- To simplify SQL Server's logging logic. When an operation was logged, earlier versions of SQL Server had to evaluate numerous criteria to determine the correct logging. In SQL Server 2000, all statements are logged based on the recovery model; it is no longer statement specific (although the performance gains are only for a select number of specific statements).
- To determine whether or not up-to-the-minute recovery is required.

Full Recovery Model The Full recovery model requires SQL Server to log every operation in full. This means that every operation will have rows written to the transaction log that allow the database to be recovered to any point in time and with no work loss exposure in the event of a database failure where the log is still accessible. This is based on a solid backup strategy, but if you are running in the Full recovery model, you have the most options available to you. It is also the best model to be in at the time of a failure (if data is changing—read-only databases have a few other options).

In the Full recovery model, no operations run with minimal logging. In fact, some operations might not perform as they did in SQL Server 7.0. This is a common source of misunderstanding, especially for people upgrading from SQL Server 7.0 to SQL Server 2000. Batch operations, which performed minimally logged operations (for example, building or rebuilding indexes), require more log space and take longer in SQL Server 2000 than they did in SQL Server 7.0. If you want to run in a minimally logged mode, you can change your recovery model to Bulk-Logged; however, there are some important trade-offs of which you should be aware.

Who Should Use the Full Recovery Model? The Full recovery model is the only recovery model that has no work loss exposure as long as the continuity of the log is not broken and the transaction log is accessible at the time of failure. Therefore, if your databases are processing transactions at all times and every transaction should be recoverable, this is the only recovery model you should consider. The Full recovery model is the one that can provide all recovery options, and it also has the least potential for data loss. Both point-in-time recovery and up-to-the-minute recovery are possible only when the database recovery model is set to Full.

Bulk-Logged Recovery Model The Bulk-Logged recovery model allows certain operations to run more efficiently than the Full recovery model because it minimally logs certain operations. Instead of logging every operation fully, the Bulk-Logged recovery model only logs the extents modified during the operation. This keeps the active log small and might allow you to have a smaller defined transaction log size than the Full recovery model. To be able to recover the operation, the transaction log should be backed up immediately on the completion of any Bulk-Logged operation, and this is true for any of the recovery models.

When the transaction log is backed up in this mode there are two steps. First—and this is the big difference for the Bulk-Logged model—SQL Server backs up all of the extents modified by the bulk operations performed (the specific commands defined by "bulk" are listed later). Second, the transaction log is backed up as it would be during a log backup in the Full recovery model. This is similar in concept to how a differential backup works, but the extents backed up are only those changed by the bulk operation. This allows some operations to occur quickly and with minimal logging (only a bitmap is maintained through the operation), but your recovery options are limited. First, if you have performed a bulk operation, this transaction log backup does not allow point-in-time recovery during a restore. Second, if the data portion of the database is not accessible (because, for example, the disks failed), a transaction log backup is not possible after a bulk operation has occurred and you are running in Bulk-Logged recovery model. The following bulk operations are minimally logged in this recovery model:

- Index creation or rebuilds
- Bulk loading of data (fast load) including (but not limited to) BULK INSERT, Data Transformation Services (DTS) Bulk Load, and *bcp*
- SELECT INTO when creating permanent tables
- WRITETEXT and UPDATETEXT for binary large object (BLOB) manipulation

Technically, you can still have point-in-time and up-to-the-minute recovery when running in Bulk-Logged recovery model, but this is possible only when bulk logged operations have not occurred since the last transaction log backup. However, this can create confusion and the process is error-prone. Instead of running in Bulk-Logged recovery model all the time, change between recovery models as part of your batch processes. If you are in control of the recovery models, you can force transaction log backups to occur at the most appropriate times, minimizing the potential for data loss.

It is important that you perform log backups immediately after a batch operation to ensure that everything is recoverable. Consider this timeline:

- 12:00 A.M.—Transaction log backup occurs (transaction log backups occur hourly).
- 12:10 A.M.—Batch operation begins.
- 12:20 A.M.—Batch operation completes.
- 12:47 A.M.—Database becomes suspect due to drive failure.
- 12:50 A.M.—You become aware of the suspect database. You attempt to access the tail of the transaction log, but you receive the following errors:

```
Server: Msg 4216, Level 16, State 1, Line 1
Minimally logged operations cannot be backed up when the database is
unavailable.
Server: Msg 3013, Level 16, State 1, Line 1
BACKUP LOG is terminating abnormally.
```

On the CD If you would like to see this process and log backup failure, use the script Cannot_Backup_Tail_After_Bulk_Operation.sql to test.

At 12:50 A.M., all you can do is restore the database and the logs up until 12:00 A.M. If you had backed up the log at 12:20 A.M., your database would not have been in a bulk logged state (regardless of the recovery model setting to Bulk-Logged). You can back up the tail of the transaction log when you are running in Bulk-Logged recovery model only if no bulk operations have occurred. By backing up the transaction log immediately after a bulk operation you are in effect resetting the bulk logged state such that transaction log backups can be performed without requiring access to the data portion of the database. If the

database had not been in a bulk logged state at 12:50, then you would have been able to get the tail of the transaction log. If the tail of the log had been accessible, you would have up-to-the-minute recovery and no data loss. Instead you have lost *all* activity since 12:00 A.M.

To take these concepts further, look at another scenario. What if the database were to become corrupt at 12:15 A.M. in the middle of the batch operation? You know that the tail of the transaction log is not accessible because you are in the process of a bulk operation in the Bulk-Logged recovery model. However, your data loss is everything past 12:00 A.M. You certainly could have prevented some—and possibly all—of this data loss. Performing a transaction log backup at 12:10 A.M. when the database was accessible (right before the bulk operation began) would have at least brought you up to 12:10 A.M., the moment prior to the bulk operation. If the bulk operation were the only operation occurring from 12:10 A.M. to 12:15 A.M. (when the database became corrupt), the transaction log backup could be used to bring the database up to 12:10 A.M. Once recovered to 12:10 A.M., the bulk operation could be executed again to bring the database up to the time of the failure and continue it moving forward.

It is critical to back up your transaction log both immediately before performing batch operations and immediately after performing a batch operation. Both minimize the overall potential for data loss in the event of a failure. Remember that if the database is set to the Bulk-Logged recovery model *and* you have performed a bulk operation, you cannot backup the tail of the log even if the transaction log file is accessible. If you have not performed a bulk operation, you can back up the log. For this reason, some people might consider always running in the Bulk-Logged recovery model. However, this can be dangerous because you are no longer entirely in control of the recovery. Bulk operations are not necessarily limited to only DBAs or system administrators. Anyone who owns a table can create or rebuild indexes of their tables, anyone with Create Table permissions can use SELECT INTO to create a permanent table, and anyone who has access to text data can manipulate it with WRITETEXT and UPDATETEXT. Because of this, it is very important to know and limit when operations are logged fully or minimally. If you are responsible for data recovery and your environment cannot afford data loss, the only way to minimize data loss is by running in the Full recovery model and controlling changes to the Bulk-Logged recovery model. Only when the Bulk-Logged recovery model is appropriate should you switch. In some environments it might not even be possible to switch. The best practice, if you determine that it is acceptable to periodically change to the Bulk-Logged recovery model, is to change within batch processes. This practice ensures that the window of potential work loss is limited to only appropriate times of day.

Who Should Use the Bulk-Logged Recovery Model? If your databases are not processing transactions around the clock or if you are willing to have work loss exposure to achieve better performance of your batch operations, you might consider a temporary change to the Bulk-Logged recovery model. However, even if you decide that this is acceptable, you should change to Bulk-Logged during the batch operation (preceding the switch with a log backup) and then change back when the operation is complete (following the switch with another log backup). Again, these operations only protect data and minimize the potential window for data loss. For batch processes, an example is given to completely detail the process to optimally change recovery models in the section "Changing Between Recovery Models" later in this chapter.

Also, as a secondary consideration, depending on the length of the bulk operation, you might consider trying to break down large or complex batch operations that might cause the transaction log to grow excessively large. In fact, to minimize the potential for data loss (because you cannot back up the tail of the log if the database becomes suspect), you might perform log backups during the batch process and between some of the steps of the bulk operations. Breaking down any very large or complex operations and performing log backups between the larger steps allow more recovery options.

Simple Recovery Model The Simple recovery model logs data as if the database were in the Bulk-Logged recovery model; however, the log is periodically cleared. Instead of keeping all of the log information until a transaction log backup is performed, SQL Server clears the log information from the transaction log as the data is synchronized from memory to its appropriate location on disk (at checkpoint). Because the data no longer resides solely in memory (for the data portion of the database) there is no need for the information to be stored in the transaction log for automatic recovery (remember, this is SQL Server's primary reason to have a transaction log). However, if the transaction log is periodically cleared, transaction log backups are not possible.

The Simple recovery model is the easiest recovery model to use, as the log is cleared periodically and automatically. In the Simple recovery model, administration is simple because no transaction log maintenance is required (the transaction log is maintained through log truncation when a checkpoint occurs). In fact, only two backup types are possible: full database backups and differential database backups. However, this simplified administration comes at the expense of significant work loss if the database becomes suspect. In fact, you can recover only up to your last full database backup or your last differential database backup.

Who Should Use the Simple Recovery Model? If your databases are periodically built as copies of data from other transaction processing systems and

can be rebuilt if necessary, you might consider the Simple recovery model with a full database backup on completion of the database build. The Simple recovery model is common for predominantly read-only or development and test databases for which up-to-the-minute recovery is not necessary and data loss is not critical to the success of the business.

Choosing the Right Recovery Model: An Example Test Case To show a quick overview of the effects on the database's transaction log size (meaning the amount of space required to "log" the operation), the size of the transaction log backup, and the speed of the operation, a simple test was performed using a very specific operation: SELECT INTO (see Table 9-2). SELECT INTO creates a new table called TestTable based on a table called a charge table from another database. The charge table has 800,000 rows and the data is roughly 40 MB in size.

Table 9-2 SELECT INTO Operation

Database Recovery Model	Duration (Seconds)	Database Transaction Log Size	Transaction Log Backup Size
Simple	8.5	< 4 MB	Not allowed
Bulk-Logged	8.5	< 4 MB	~40 MB
Full	14	~40 MB	~40 MB

The interesting observations come from the fact that Simple and Bulk-Logged seem to have the same performance and the same active log size. However, recovery models do not affect all operations. In the second test a single update is performed against all 800,000 rows in TestTable. This caused the transaction log for all three databases to grow significantly to handle the modification and there was no difference in the operation's duration or the size of the transaction log (where a transaction log backup is permitted).

Table 9-3 UPDATE Operation

Database Recovery Model	Duration (Seconds)	Database Transaction Log Size	Transaction Log Backup Size
Simple	18	230 MB	Not allowed
Bulk-Logged	18	230 MB	~54 MB
Full	18	230 MB	~54 MB

From an interpretation of Tables 9-2 and 9-3, you might think that the best recovery model to use is the Bulk-Logged recovery model, because it seems to allow transaction log backups and because the operations affected by recovery models run faster. However, you are missing a key element, because the transaction log is not always available for a transaction log backup when running the Bulk-Logged recovery model. If the device on which the data resides is not available when a transaction log backup is attempted, then a transaction log backup cannot be performed, resulting in data loss. Up-to-the-minute recovery is not always possible with the Bulk-Logged recovery model. Getting familiar with the different recovery models and their trade-offs is very important for production databases, as the recovery model can affect speed, logging, and recovery.

On the CD The code for these two operations and tests can be found in script file Recovery_Model_Log_Sizes.sql. The code for this script might need numerous alterations to run on your test server. Be sure to carefully read all comments.

Changing Between Recovery Models

Generally, you will choose a recovery model and stick with it. However, in some cases you might want to switch between two recovery models and then switch back. For example, some databases run in Full recovery model most of the time and switch to Bulk-Logged during bulk loading or batch processing. As long as you realize the potential work loss exposure if your database were to become suspect and you have decided it is acceptable, then changing between the recovery models is reasonable. There are six possible combinations as shown in Table 9-4, but only two are common. In fact, the recovery model is usually only changed for certain operations and then returned after the operation completes. However, changes to the simple recovery model should be performed with caution, as this change breaks the continuity of the transaction log.

Table 9-4 Changing Recovery Models and Impact on Backups

TO \ FROM	Simple	Bulk-Logged	Full
Simple	N/A	Nothing required. A transaction log backup is recommended before the switch.	Nothing required. A transaction log backup is recommended before the switch.
Bulk-Logged	New recovery set required; full database or differential database backup or a complete file or filegroup backup must be performed.	N/A	Nothing required. A transaction log backup is recommended before the switch.
Full	New recovery set required; full database or differential database backup or a complete file or filegroup backup must be performed.	Nothing required. A transaction log backup is recommended after the switch.	N/A

The most common changes occur from the Full recovery model to the Bulk-Logged recovery model and then back to the Full recovery model again. In fact, the most common strategy actually includes steps to minimize the window of potential data loss by performing the recovery model changes and the transaction log backups as part of the batch process. The following is a high-level overview of the steps that should be performed when switching between full and Bulk-Logged recovery models.

On the CD If you would like to see sample code, including various optional parameters to each of these commands, use the script DB_Alter_For_Batch_Operation.sql.

- Database is currently in the Full recovery model.
- As part of the bulk operation's batch process, perform a transaction log backup immediately before changing to the Bulk-Logged recovery model.
- Change the recovery model to Bulk-Logged using the ALTER DATABASE command:

```
ALTER DATABASE SET RECOVERY BULK_LOGGED
```

- Perform your bulk operations, data loads, index builds or rebuilds, and so on.
- Change the recovery model back to Full again using the ALTER DATABASE command:

```
ALTER DATABASE SET RECOVERY FULL
```

- Immediately after the change, perform a transaction log backup.

By performing a transaction log backup before as well as after the bulk operation, you ensure that the window for potential data loss is significantly reduced. As long as the transaction log backup occurs, you have something from which you can recover during manual recovery. However, you cannot back up the tail of the log if the database were to become damaged during the bulk operation. If there are any user transactions that are not recoverable by rerunning your bulk operation, you should remain in the Full recovery model instead of changing to the Bulk-Logged recovery model. You will lose these transactions if the database becomes inaccessible.

Recovery models have an obvious and significant impact on the available recovery options. Knowing how the environment is defined and actively choosing your recovery model is the first step in a complete and effective recovery strategy. In the next section you will combine your understanding of recovery models with each backup type to determine the specific database requirements necessary to support your downtime and data loss specifications.

Backup Types

You always plan your backup strategy based on your desired recovery abilities. A recovery-oriented strategy is always best. For example, if you want up-to-the-minute recovery you must use the Full recovery model. How do you actually achieve an up-to-the-minute recovery? Which backups are critical and in what order? What could go wrong? Are there operations, settings, or user actions that could negatively affect the database recovery? In this section, all of the backup

types are discussed, and more important, the common combinations that yield effective strategies to offer the most options for recovery are debated. Some plans even include redundancy within the backup strategy (yes, a backup plan in the backup plan!).

SQL Server 2000 (as well as SQL Server 7.0) offers seven different backup types: full database backups, transaction log backups, differential database backups, full file backups, full filegroup backups, differential file backups, and differential filegroup backups. Each one of these is often called something similar in other documentation and articles, and even the authors of this book have probably used multiple terms. For consistency and clarity in this chapter, only the terms listed in Table 9-5 are used. However, when you read other documentation you might see slightly different names used for each of these backup types.

Table 9-5 Backup Naming Conventions

Naming Conventions Used in This Chapter	Naming Conventions Commonly Used Elsewhere or Important Notes About Usage
Full database backup	Database—complete, complete database, full database, database backup
Differential database backup	Database—differential, differential backup
Transaction log backup	Log backup, tranlog backup, T-log backup, transaction backup
Full file backup	File backups
Differential file backup	File differentials
Full filegroup backup	Filegroup backups
Differential filegroup backup	Filegroup differentials
Full file/filegroup backups	Applies to both full file backups and full filegroup backups
Differential file/filegroup backups	Applies to both differential file backups and differential filegroup backups
File/filegroup backups	Applies to all types of file/filegroup backups: full file/filegroup backups and differential file/filegroup backups
Full backups	Applies to all three: full database backups, full file backups, and full filegroup backups
Differential backups	Applies to all three: differential database backups, differential file backups, and differential filegroup backups

Full Database Backups

A full database backup is the most complete backup you can create. Because it is the easiest recovery strategy to manage, it is the foundation for the most commonly used recovery strategy. Although full database backups are complete, they are often not used alone. Typically full database backups are used in conjunction with other transaction log backups and differential database backups.

> **Tip** A full database backup is the most common backup type used to define the starting point of a recovery strategy, but it is not the only starting point. In fact, it is not even necessary that you ever perform a full database backup. There are really two starting points for your recovery process: full database backups or file/filegroup backups.

A full database backup is complete in that everything necessary to access the data of that database—even the database structure—is backed up and the information about the backup (what was backed up, when it was backed up, what the database structure looks like, and what type of backup it is) can be easily queried from backup devices. On the restore, you do not need to create the database prior to restoring a full database backup; instead you can let the restore create and even move the database files to other locations. When a full backup is performed it is optimized to back up only pages (rather extents) that contain data, so the size of a backup is the size of the database minus the unallocated space. When a full database backup is performed, the image created in the backup is an image of the database at the backup's completion.

How Do Full Database Backups Work? SQL Server needs a way to access data quickly to generate an image of the database while users are actively processing and create a backup that can restore a transactionally consistent database to the way it looked on the completion of a backup. This process is quite logical. Figure 9-5 shows a graphical representation of the logical sequence of steps for the full database backup process.

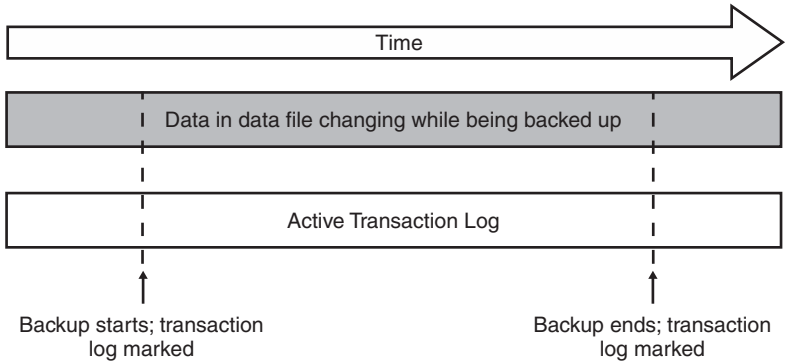

Figure 9-5 Full backups do not block active transactions.

The logical sequence of steps is as follows:

1. The first step is to perform a database checkpoint. Remember, the purpose of a checkpoint is to write all dirty pages to disk. A dirty page is one that has changed since it was brought into memory, regardless of the state of the transaction that modified it. The checkpoint is performed to batch the writes to disk and simplify Automatic Recovery, the process that SQL Server goes through on startup to make sure that each database is transactionally consistent. By performing a checkpoint, SQL Server guarantees that what is in the data portion of the database is as close to current as possible, even if the transactions are still processing.
2. Mark the transaction log. A marker is put into the transaction log to define where the backup began. This is used later in the backup process.
3. Read from data files. In this step SQL Server reads the image of data from the data files. Even while active users are processing, SQL Server continues to use only the pages that are already on disk, even if they are not logically consistent with the processing transactions. This does not seem like it would work, but it does! The result of this phase is considered a logically inconsistent (often called fuzzy) set of pages.
4. Again, mark the transaction log. This marker defines where the backup completed.

5. Back up the interesting part of the transaction log. By using the two markers created by Steps 2 and 4 and some additional information about the earliest transaction in the log (if it is earlier than the first marker), SQL Server is able to get directions for how to update the logically inconsistent image of data that has been backed up (although this only needs to be used during a restore). The transaction log backup that is performed is not a typical log backup, per se. The log is only backed up. The log is not cleared (this is the default setting for a transaction log backup). In fact, a full database backup does not touch nor does it break the continuity of the transaction log.

The end result of this sequence is that active transactions *never* wait and are not negatively impacted by a full database backup. It is true that the system will see a heavier I/O load, but if the backup devices and database files are well balanced and placed properly, the impact of the full database backup on actively processing users is minimal.

If full database backups are so optimal, why not perform them constantly? It might seem logical to begin another full database backup once one completes. This, in fact, is not typically a good idea.

> **Note** Certain storage-assisted strategies exist and can successfully perform full backups in a rolling fashion. This is even recommended for some situations. However, without storage-assisted backups, rolling full database backup after full database backup is not recommended.

Why? Full database backups conflict with other operations. In fact, while a full database backup is running there are other operations that will be paused until it completes. This can cause a few negative side effects. The operations and side effects with which full database backup conflicts are as follows:

- **Transaction log backups** The effects of this can be quite serious. Remember, SQL Server backs up the transaction log as part of a full backup to make the full backup transactionally consistent during restore. To back up the transaction log, the transaction log needs to be accessible and complete, meaning that SQL Server must have access to all log activity that has occurred throughout the entire full backup. For the log to be accessible, it cannot have been cleared. If

there are other operations that depend on the transaction log (for example, log shipping), log shipping will be paused until the full database backup completes. If your full database backup takes 6 hours to run, your secondary site could be 6 hours behind. Is 6 hours of data loss acceptable if you were to have site failure and lose the primary? Using full database backups as the base for your database backup strategy might not be best! There are other options.

- **Alternatives** Any operation that changes the database structure:
  - No autogrow
  - No manual growth with ALTER DATABASE
  - No autoshrink
  - No manual database shrinks with DBCC SHRINKDATABASE
  - No manual database file shrinks with DBCC SHRINKFILE

A secondary problem related to the transaction log filling, the effect on the database once the transaction log is full is that no modifications are allowed until space is available in the transaction log. However, when a full database backup is already being performed, no operations can run to clear the transaction until the full database backup has finished. Neither the data portion nor the log portion is allowed to grow during a full database backup. It is not as critical that the data portion cannot autogrow, as this should be relatively unlikely with proper capacity planning. However, if you decide to use full database backups as the base for your backup strategy, make sure you have an adequately sized transaction log—as large as necessary to hold the transactions that occur over the entire time you perform full database backups. Because most database backups are performed at "off" hours during which activity is minimal, this might not present a problem for you. However, if a full database backup is performed for some other purpose (during regular or heavy-use business hours) it could present a problem.

> **Tip** Not allowing autoshrink or manual shrinks of the database should not present a problem, as these are rarely used. Autoshrink is not a typical option for production servers; it is an option more appropriate to scaled-down SQL Server databases.

Speaking of very large databases (VLDBs), does your VLDB have a large portion of data that is predominantly read-only? Do you really need complete backups of that data very often? VLDB presents additional concerns with the strategies based on full database backups. As various strategies are discussed, you will see other alternatives to frequent full database backups that are helpful for VLDBs, possibly allowing a strategy that works without ever performing a full database backup.

Transaction Log Backups

For high availability, transaction log backups are the most important type of backup. They are a critical component to any backup strategy that desires up-to-the-minute or point-in-time recovery. The more frequent your log backups are, the more likely you will have a successful recovery with minimal data loss (see Figure 9-6). A transaction log backup provides a way to capture the changes that have occurred since the last transaction log backup. Additionally, when transaction log backups occur, SQL Server clears the inactive portion of the transaction log; this helps free space within the transaction log and removes the instructions so that recovery is possible.

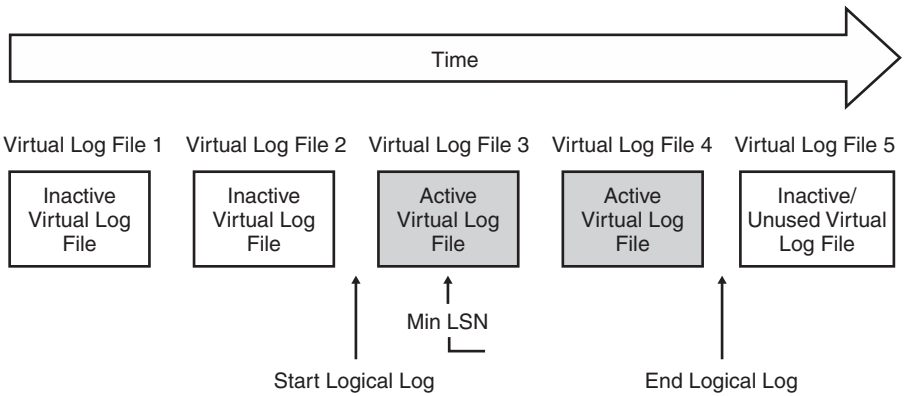

Figure 9-6 Frequent log backups minimize potential data loss.

> **Note** Clearing the log does not necessarily clear the entire log. If you review the percent log used within the System Monitor, you might never see the percent log used value for the transaction log (even immediately after a backup) drop to zero. This is based on how the transaction log backups work and is not likely a concern.

How Do Transaction Log Backups Work? Take, for example, a 100-MB transaction log divided into five VLFs (for details on VLFs review the earlier section "Understanding Database Structures"). Log activity starts at the beginning of the log and at the current time activity is midway through the fourth VLF. There is an open transaction that began in the third VLF and that is marked as the minimum log sequence number (min LSN). You have just requested a transaction log backup.

SQL Server reviews the log to determine where it is active versus where it is inactive. VLFs that do not contain any active transactions are inactive, but used VLFs. SQL Server then backs up all VLFs and clears those that are inactive. At the time of the backup, at least one VLF is always active. Depending on the size of the VLFs and the size of the open transactions, you might not be able to clear anything from your transaction log. To ensure optimal transaction log backups and optimal performance for log-related operations, it is important to make sure that you have small, efficient transactions. In fact, if you have large batch operations you might consider one of two options:

- Use the Bulk-Logged recovery model (if the work loss exposure is acceptable and the operation has performance and logging advantages when running in the Bulk-Logged recovery model).
- Break the batch operations into smaller, more manageable chunks, if possible, and perform log backups as part of the batch operation.

If all transactions and operations are relatively small, this allows more effective management of the transaction log, keeping it small and easily recoverable.

The Effects of Recovery Models on the Transaction Log Effects on the transaction log made by the recovery model are evident in many areas: performance, active log size versus transaction log backup size, and whether or not log backups are allowed. In the simple recovery model, transaction log backups are not allowed. In the Bulk-Logged recovery model the active log is much smaller than the transaction log backup because SQL Server uses a bitmap to keep track of changed extents during bulk operations. This keeps the active log size small, but causes the transaction log backup size to be as large as a transaction log backup performed in a database running with the Full recovery model. For databases running in the Full recovery model, the transaction log backup should be close to the size of the current transaction log space used. To estimate the size of the transaction log backup, use DBCC SQLPERF (logspace) or

directly query the master.dbo.sysperfinfo table. For example, to see the "used" size of the transaction log for a database in the Full recovery model, use the following query prior to performing a backup:

```
SELECT * FROM master.dbo.sysperfinfo
WHERE object_name = 'SQLServer:Databases'
AND counter_name = 'Log File(s) Used Size (KB)'
AND instance_name = 'YourDatabaseName'
```

This query is not helpful for estimating the size of the transaction log backup for a database running in the Bulk-Logged recovery model. This cannot be estimated and the transaction log backup could be as large as the database if there has been a significant amount of Bulk-Logged activity. Make sure to test transaction log backup sizes during testing and development and be sure to plan for (in terms of backup device location) much larger transaction log backups than the database's transaction log size.

> **More Info** Using sysperfinfo for monitoring will be covered in Chapter 15, "Monitoring for High Availability."

Differential Database Backups

A new feature introduced with SQL Server 7.0, differential database backups specifically minimize backup and recovery time. Differential database backups reduce recovery time by allowing databases with somewhat isolated activity to have only the changed extents backed up. When a differential backup is performed, SQL Server goes through the same process as a full database backup. However, the backup focuses on only the extents that were changed since the last full database backup. Differential backups can be performed in any database at any time, *even after* the continuity of the transaction log has been broken. However, a differential backup—as does a full database backup—pauses transaction log backups until it has completed. Instead of requiring a full database backup, a differential database backup allows you a faster way to back up your database and protect your recovery process, not relying so heavily on transaction log backups or complete backups of the entire database.

The recovery of a database using differential database backups is similar to applying the sequence of log backups that occurred over the same period of time. However, it is significantly faster. For all work that has been committed prior to the start of the differential database backup, there are completed images of those pages, even if a page was changed multiple times. The differential database

backup is almost like a "mini" full database backup, but of only those extents that have changed. After the extents are restored, the transaction log of activity that occurred while the differential database backup was being performed is applied. As with the full database backup, the differential database backup restores a transactionally consistent image of the database as it looked at the completion of the differential database backup instead of at the beginning. This results in a faster recovery sequence than if using only transaction log backups. Instead of applying all of the individual changes that led up to a specific point in time, the differential database backup already includes the end result of all of the changes. Differential backups can be a key element of any backup strategy focused on high availability, as recovery time must be optimized.

Should every database use the differential backup strategy? Differential database backups are best in cases in which activity is somewhat isolated or in which the database has a large amount of relatively static data. If almost every page changes within your database during the expected time between the differential backups or if you have the ability to perform full database backups frequently because there are no negative issues associated with costs (for example, backup media costs, administrative costs, and recovery costs in terms of the operations that are paused), then performing full database backups instead of differential database backups can lead to an even shorter recovery time because fewer backups need to be restored. Differential database backups have the same conflicts with changes in database structures or transaction log backups as do full database backups.

How Differential Database Backups Work To keep track of which extents have changed within the database, there is an internal bitmap within each file. Every time the information within the file is backed up completely, as with a full database backup, the bitmap is reset. This is very efficient because only one bit is used per extent. Therefore one 8-K page maps to approximately 4 GB of data. Even in larger files, there is only one bitmap per 4 GB and they are a doubly linked list starting with the first.

> **Note** The bitmap is new to SQL Server 2000. When a differential database backup was performed, SQL Server 7.0 would scan the entire database, reviewing the header of every page to see if it had been modified since the last full database backup. This was extremely inefficient when the backup was performed, but had no effect on the restore. The restore was as efficient as it is in SQL Server 2000. SQL Server 7.0 was optimized only for restore. SQL Server 2000 is optimized for both backup and restore.

A differential database backup is therefore very similar to a full database backup, except that it is only the extents that have changed; therefore the backup is usually significantly smaller. However, because the bitmap is only reset when the file is completely backed up—as with a full database backup—the size of the differential database backups might approach the size of a full database backup over time or if an operation is performed that affects most of the pages of the database.

> **Note** There are some caveats to this if you combine full database backups and differential database backups with full file/filegroup backups and differential file/filegroup backups. The differential bitmap, which is file based, gets reset anytime the full database or the full file/filegroup is backed up. Therefore, if they are combined, you can end up with a significantly slower backup to determine the differential. However, combining file, filegroup, and full backups for the same database is uncommon and therefore not recommended.

Even if you choose to add differential database backups to your recovery strategy, it is important that you also continue to periodically perform full database backups as part of a complete backup strategy. By performing full database backups you will reset the bitmap and have a more recent backup (this is sometimes good because of poor media quality).

Full File/Filegroup Backups

Both full file backups and full filegroup backups work in the same way that a full database backup works in terms of the data that is backed up. However, full file/filegroup backups do not back up the transaction log as part of the backup. Instead, you are required to back up the transaction log separately to make the full file/filegroup backup consistent at the time of restore. This means two things: you must be using either the Full or Bulk-Logged recovery model and log backups are never paused. This is one of the best choices for systems using log shipping or systems with large databases. It is excellent for log shipping because log backups will never have significant delays due to full database backups. It is also ideal for VLDBs because you can set different frequencies for the different files (and usually different types of data) within your database. If your VLDB has a large portion of read-only data that only needs to be backed up monthly, then you can back up the file or files in which that data resides only monthly.

Why should some implement their backups based on file/filegroup strategies and others not consider it? The reason is simple: implementing a file/filegroup backup strategy adds a significant amount of administrative complexity. You must design your database with file and filegroup strategies in mind, and restoring the proper combination of backups is also more challenging.

Differential File/Filegroup Backups

Both differential file backups and differential filegroup backups work in the same way that a differential database backup works. When the differential file or filegroup backup is performed, all of the extents that changed since the last full file or filegroup backup, respectively, will be backed up. The file-based bitmap is reset each time a full file backup or a full filegroup backup is performed. This is the only part about these that can be tricky. If you were to perform a full filegroup backup and then a differential filegroup backup, the differential filegroup backup would include only those extents that have changed, and it would use the bitmap to do so. If you start combining differential filegroup backups with differential file backups with differential database backups, the bitmaps might get reset. If the bitmap is reset, SQL Server needs to walk the entire database to ensure that all extents are properly backed up. This does not impact the restore, but it will be a less efficient backup. If you choose a file/filegroup-based backup strategy, you should be consistent with the levels at which you perform differential backups. This ensures not only a speedy restore, but also an efficient backup.

> **Tip** It is important to realize that even when filegroups are used for read-only activity, SQL Server 2000 requires that transaction log backups be applied during recovery to make the file or filegroup consistent, even when no data has changed. It is important to perform file/filegroup differential backups on the file/filegroups that are logically read-only. During the restore, you can use the last full file/filegroup backup, the last file/filegroup differential backup, and all of the transaction logs from there. The differential backups will be effectively empty, but it significantly reduces the number of transaction logs that must be applied and therefore reduces the amount of time it takes to recover.

Summary

This chapter provides you with the foundation you need as you start to understand backup and restore better. You must know how SQL Server works behind the scenes if you are to put together a backup and restore plan that fits the needs of your company. That means knowing what is possible, when it can happen, and what you should do if something does not go as planned. Without this understanding, you will have a hard time putting together a cohesive disaster recovery plan.