

MS SQL SERVER

Managing log files in MS SQL server

How to manage a large log file in MS SQL Server

Overview

Introduced in SQL Server 2000 was the ability to shrink the physical size of database data and transaction log files. This can be very useful under special circumstances, for example after archiving data to some other location. However, we often see DBA's doing this on a regular basis and the purpose of this article is to explain some of the downsides of shrink and what actually happens when you shrink a database file.

More information

You can shrink a database file using either DBCC SHRINKDATABASE (which targets all files for the database) or DBCC SHRINKFILE (which targets a particular database file). I prefer SHRINKFILE. I will not go through the details of the commands here; they are documented in SQL Server Books Online. Let us first determine what actually happens when you shrink a database file:

Shrinking of data file

When you shrink a data file, SQL Server will first move pages towards the beginning of the file. This frees up space at the end of the file and the file can then be shrunk (or as I like to view it: "cut off at the end").

Shrinking of transaction log file

SQL Server cannot move around log records toward beginning of the file as each record which is moved would need to be logged and we'd be in a catch-22 situation. So, SQL Server can only cut down the file size if the file is empty at the end of the file. The end-most log record sets the limit of how much the transaction log can be shrunk. A transaction log file is shrunk in units of Virtual Log Files (VLF). You can actually see the VLF layout using the undocumented DBCC LOGINFO command, which returns one row per virtual log file for the database:

DBCC LOGINFO('myDatabase')

| Field | FileSize | StartOffset | FSeqNo | Status | Parity | CreateLSN |
|-------|----------|-------------|--------|--------|--------|------------------|
| 2 | 253952 | 8192 | 11 | 0 | 128 | 0 |
| 2 | 253952 | 262144 | 13 | 0 | 128 | 0 |
| 2 | 270336 | 516096 | 12 | 0 | 128 | 7000000025000288 |
| 2 | 262144 | 786432 | 14 | 2 | 128 | 9000000008400246 |

The interesting column is "Status". 0 means that the VLF is not in use and 2 means that it is in use. In my example, I have 2 at the end of the file (read the result in StartOffset order) and this means that the file cannot currently be shrunk. In 7.0, you have to generate dummy transactions so that the usage of the log file wraps toward the beginning of the file. You can then empty the log file using BACKUP LOG and then shrink the file. In SQL Server 2000, the generation of dummy log records is done for you when you execute the DBCC SHRINKFILE command.

So what is the problem? Why shouldn't I shrink database files on a regular basis?

Have a look at below list and then you can determine for yourself whether you want to shrink database files regularly or not:

- Each page moved will be logged to the transaction log. Say you have a database using 3GB of data and indexes, then the log file will need 3GB space for the shrink. This 3GB will also be included in the following transaction log backup. (Applies to shrinking of data files.)
- After the shrink, as users add rows etc in the database, the file has to grow again. Growing database files are expensive operations. I.e., it hurts performance. During the grow operation, some modifications will be blocked until the grow operation has finished. (Applies to shrinking of both data and log files.)
- There are situations where autogrow doesn't "catch up" with the space usage requirements. This will result in an error message from SQL Server when the modification is performed, returned to the client application: error 1105 if data is full and 9002 if log is full. (Applies to shrinking of both data and log files.)
- Moving datapages around will fragment your database. (Applies to shrinking of data files.)
- Heavy shrinking and growing of database files will fragment your file system, which will hurt performance even more. (Applies to shrinking of both data and log files.)

Conclusion

Only you can determine in the end whether you want to shrink or not. With above information, you hopefully have enough to go on when taking that decision.

References

Below are KB articles regarding size of transaction log file and shrinking of the transaction log file.

[Log file filling up](#)

[Considerations for Autogrow and AutoShrink](#)

INF: Shrinking the Transaction Log in SQL Server 2000 with DBCC SHRINKFILE

1. You must run a BACKUP LOG statement to free up space by removing the inactive portion of the log.
2. You must run DBCC SHRINKFILE again with the desired target size until the log file shrinks to the target size.

The following example demonstrates this with the pubs database and attempts to shrink the pubs_log file to 2 MB:

1. Run this code:

```
DBCC SHRINKFILE(pubs_log, 2)
```

NOTE: If the target size is not reached, proceed to the next step.

2. Run this code if you want to truncate the transaction log and not keep a backup of the transaction log. Truncate_only invalidates your transaction log backup sequence. Take a full backup of your database after you perform backup log with truncate_only:

```
BACKUP LOG pubs WITH TRUNCATE_ONLY
```

-or-

Run this code if you want to keep a backup of your transaction log and keep your transaction log backup sequence intact. See SQL Server Books Online topic "BACKUP" for more information:

```
BACKUP LOG pubs TO pubslogbackup
```

3. Run this code:

```
DBCC SHRINKFILE(pubs_log,2)
```

The transaction log has now been shrunk to the target size.

INF: Transaction Log Grows Unexpectedly or Becomes Full on SQL Server

In SQL Server 7.0 and SQL Server 2000, with the autogrow setting, transaction log files can expand automatically.

Typically, the size of the transaction log file stabilizes when it can hold the maximum number of transactions that can occur between transaction log truncations that either checkpoints or transaction log backups trigger.

However, in some situations the transaction log may become very large and run out of space or become full. Typically, you receive the following error message when a transaction log file takes up the available disk space and cannot expand any more:

Error: 9002, Severity: 17, State: 2
The log file for database '%.*s' is full.

In addition to this error message, SQL Server may mark databases suspect because of a lack of space for transaction log expansion. For additional information about how to recover from this situation, see the "Insufficient Disk Space" topic in SQL Server Books Online.

Additionally, transaction log expansion may result in the following situations:

- A very large transaction log file.
- Transactions may fail and may start to roll back.
- Transactions may take a long time to complete.
- Performance issues may occur.
- Blocking may occur.

Causes

Transaction log expansion may occur because of the following reasons or scenarios:

- [Uncommitted Transactions](#)
- [Extremely Large Transactions](#)
- [Operations: DBCC DBREINDEX and CREATE INDEX](#)
- [While Restoring from Transaction Log Backups](#)
- [Client Applications Do Not Process All Results](#)
- [Queries Time Out Before a Transaction Log Completes the Expansion and You Receive False 'Log Full' Error Messages](#)
- [Unreplicated Transactions](#)

Uncommitted Transactions

Explicit transactions remain uncommitted if you do not issue an explicit COMMIT or ROLLBACK command. This most frequently occurs when an application issues a CANCEL or a Transact SQL KILL command without a corresponding ROLLBACK command. The transaction cancellation occurs, but it does not roll back; therefore, SQL Server cannot truncate every transaction that occurs after this because the aborted transaction is still open.

You can use the DBCC OPENTRAN Transact-SQL reference to verify if there is an active transaction in a database at a particular time. For additional information about this particular scenario, click the article number below to view the article in the Microsoft Knowledge Base:

[295108](#) INF: Incomplete Transaction May Hold Large Number of Locks and Cause Blocking

[171224](#) INF: Understanding How the Transact-SQL KILL Command Works

Additionally, see the "DBCC OPENTRAN" topic in SQL Server Books Online.

Scenarios that may result in uncommitted transactions:

- An application design that assumes that all errors cause rollbacks.
- An application design that does not completely take into account SQL Server behavior when it rolls back to named transactions or specially-nested named transactions. If you try to roll back to an inner-named transaction, you receive the following error message:

Server: Msg 6401, Level 16, State 1, Line 13 Cannot roll back InnerTran. No transaction or savepoint of that name was found.

After SQL Server generates the error message, it continues to the next statement. This is by design. For more information, see the "Nested Transactions" or "Inside SQL Server" topic in SQL Server Books Online.

Microsoft recommends the following when you design your application:

- Only open one transaction unit (consider the possibility that another process may call yours).
 - Check @@TRANCOUNT before you issue a COMMIT, a ROLLBACK, a RETURN, or a similar command or statement.
 - Write your code with the assumption that another @@TRANCOUNT might "nest" yours and plan for the outer @@TRANCOUNT to be rolled back when an error occurs.
 - Review savepoint and mark options for transactions. (These do not release locks!)
 - Perform complete testing.
- An application that permits user interaction inside transactions. This causes the transaction to remain open for a long time, which causes blocking and transaction log growth because the open transaction cannot be truncated and new transactions are added to the log after the open transaction.
 - An application that does not check @@TRANCOUNT to verify that there are no open transactions.
 - Network or other errors that close the client application connection to SQL Server without informing it.
 - Connection pooling. After worker threads are created, SQL Server reuses them if they are not servicing a connection. If a user connection starts a transaction and disconnects before committing or rolling back the transaction, and a connection thereafter reuses the same thread, the previous transaction still stays open. This situation results in locks that stay open from the previous transaction and prevents the truncation of the committed transactions in the log, which results in large log file sizes. For additional information about connection pooling, click the article number below to view the article in the Microsoft Knowledge Base:
[164221](#) INFO: How to Enable Connection Pooling in an ODBC Application

Extremely Large Transaction

Log records in the transaction log files are truncated on a transaction-by-transaction basis. If the transaction scope is large, that transaction and any transactions started after it are not removed from the transaction log unless it completes. This can result in large log files. If the transaction is large enough, the log file might use up the available disk space and cause the "transaction log full" type of error message such as Error 9002. For additional information about what to do when you receive this type of error message is provided in the "More Information" section in this article. Additionally, it takes a lot of time and SQL Server overhead to roll back large transactions.

Operations: DBCC DBREINDEX and CREATE INDEX

Because of the changes in the recovery model in SQL Server 2000, when you use the Full recovery mode and you run DBCC DBREINDEX, the transaction log may expand significantly more compared to that of SQL Server 7.0 in an equivalent recovery mode with the use of SELECT INTO or BULK COPY and with "Trunc. Log on chkpt." off.

Although the size of the transaction log after the DBREINDEX operation might be an issue, this approach provides better log restore performance.

While Restoring from Transaction Log Backups

This is described in the following Microsoft Knowledge Base article:

[232196](#) INF: Log Space Used Appears to Grow After Restoring from Backup

If you set SQL Server 2000 to use Bulk-Logged mode and you issue a BULK COPY or SELECT INTO statement, every changed extent is marked and then backed up when you back up the transaction log. Although this permits you to back up transaction logs and recover from failures even after you perform bulk operations, this adds to the size of the transaction logs. SQL Server 7.0 does not include this feature. SQL Server 7.0 only records which extents are changed, but it does not record the actual extents. Therefore, the logging takes up significantly more space in SQL Server 2000 than in SQL Server 7.0 in Bulk-Log mode but not as much as it does in Full mode.

Client Applications Do Not Process All Results

If you issue a query to SQL Server and you do not handle the results immediately, you may be holding locks and reducing concurrency on your server.

For example, suppose you issue a query that requires rows from two pages to populate your result set. SQL Server parses, compiles, and runs the query. This means that shared locks are placed on the two pages that contain the rows that you must have to satisfy your query. Additionally, suppose that not all rows fit onto one SQL Server TDS packet (the method by which the server communicates with the client). TDS packets are filled and sent to the client. If all rows from the first page fit on the TDS packet, SQL Server releases the shared lock on that page but leaves a shared lock on the second page. SQL Server then waits for the client to request more data (you can do this by using DBNEXTROW/DBRESULTS, SQLNextRow/SQLResults, or FetchLast/FetchFirst for example).

This means that the shared lock is held until the client requests the rest of the data. Other processes that request data from the second page may be blocked.

Queries Time Out before a Transaction Log Completes the Expansion and You Receive False 'Log Full' Error Messages

In this situation, although there is enough disk space, you still receive an "out of space" error message.

This situation varies for SQL Server 7.0 and SQL Server 2000.

A query can cause the transaction log to automatically expand if the transaction log is almost full. This may take additional time, and a query may be stopped or may exceed its time-out period because of this. SQL Server 7.0 returns error 9002 in this situation. This issue does not apply to SQL Server 2000.

In SQL Server 2000, if you have the **auto-shrink** option turned on for a database, there is an extremely small time during which a transaction log tries to automatically expand, but it cannot because the **auto-shrink** function is running simultaneously. This may also cause false instances of error 9002.

Typically, the automatic expansion of transaction log files occurs quickly. However, in the following situations, it may take longer than usual:

- Growth increments are too small.
- Server is slow for various reasons.
- Disk drives are not fast enough.

Unreplicated Transactions

The transaction log size of the **publisher** database can expand if you are using replication. Transactions that affect the objects that are replicated are marked as "For Replication." These transactions, such as uncommitted transactions, are not deleted after checkpoint or after you back up the transaction log until the log-reader task copies the transactions to the distribution database and unmarks them. If an issue with the log-reader task prevents it from reading these transactions in the **publisher** database, the size of the transaction log may continue to expand as the number of non-replicated transactions increases. You can use the DBCC OPENTRAN Transact-SQL reference to identify the oldest non-replicated transaction.

For more information about troubleshooting unreplicated transactions, see the "sp_replcounters" and "sp_repldone" topics in SQL Server Books Online.

For additional information, click the article number below to view the article in the Microsoft Knowledge Base:

[306769](#) FIX: Transaction Log of Snapshot Published DB Cannot Be Truncated

[240039](#) FIX: DBCC OPENTRAN Does Not Report Replication Information

[198514](#) FIX: Restore to New Server Causes Transactions to Remain in Log

MORE INFORMATION

The transaction log for any database is managed as a set of virtual log files (VLFs) whose size SQL Server determines internally based on the total size of the log file and the growth increment in use when the log expands. A log always expands in units of whole VLFs and it can only compress to a VLF boundary. A VLF can exist in one of three states: ACTIVE, RECOVERABLE, and REUSABLE.

- **ACTIVE:** The active portion of the log begins at the minimum log sequence number (LSN) that represents an active (uncommitted) transaction. The active portion of the log ends at the last-written LSN. Any VLFs that contain any part of the active log are considered active VLFs. (Unused space in the physical log is not part of any VLF.)
- **RECOVERABLE:** The portion of the log that precedes the oldest active transaction is only necessary to maintain a sequence of log backups for recovery purposes.
- **RESUABLE:** If you are not maintaining transaction log backups, or if you already backed up the log, SQL Server reuses VLFs before the oldest active transaction.

When SQL Server reaches the end of the physical log file, it starts reusing that space in the physical file by issuing a CIRCLING BACK operation to the beginning of the files. In effect, SQL Server recycles the space in the log file that is no longer necessary for recovery or backup purposes. If a log backup sequence is being maintained, the part of the log before the minimum LSN cannot be overwritten until you back up or truncate those log records. After you perform the log backup, SQL Server can circle back to the beginning of the file. After SQL Server circles back to start writing log records earlier in the log file, the reusable portion of the log is then between the end of the logical log and active portion of the log.

For additional information, see the "Transaction Log Physical Architecture" topic in SQL Server Books Online. Additionally, you can see an excellent diagram and discussion of this on page 190 of "Inside SQL Server 7.0" (Soukup, Ron. Inside Microsoft SQL Server 7.0, Microsoft Press, 1999), and also in pages 182 through 186 of "Inside SQL Server 2000" (Delaney, Kalen. Inside Microsoft SQL Server 2000, Microsoft Press, 2000). SQL Server 7.0 and SQL Server 2000 databases have the options to autogrow and autoshrink. You can use these options to help you to compress or expand your transaction log.

For additional information about how these options can affect your server, click the article number below to view the article in the Microsoft Knowledge Base:

[315512](#) INF: Considerations for Autogrow and Autoshrink Configuration

There is a difference between the truncation versus the compression of the transaction log file. When SQL Server truncates a transaction log file, this means that the contents of that file (for example, the committed transactions) are deleted. However, when you are viewing the size of the file from a disk space perspective (for example, in Windows Explorer or by using the **dir** command) the size remains unchanged. However, the space inside the .ldf file can now be reused by new transactions. Only when SQL Server shrinks the size of the transaction log file, do you actually see a change in the physical size of the log file.

Causes of SQL Transaction Log Filling Up

The SQL Server transaction log can become full, which prevents further **UPDATE**, **DELETE**, or **INSERT** activity in the database, including **CHECKPOINT**. This is usually seen as error 1105:

Can't allocate space for object syslogs in database dbname because the logsegment is full. If you ran out of space in syslogs, dump the transaction log. Otherwise use ALTER DATABASE or sp_extendsegment to increase the size of the segment.

This can happen on any database, including master or tempdb. This article discusses possible causes and solutions for those problems that led to the error 1105. If your transaction log has filled and you are currently receiving error 1105, you need to empty the log by using the DUMP TRANSACTION statement. For more information about using DUMP TRANSACTION, see your SQL Server documentation.

MORE INFORMATION

A fundamental characteristic of true relational databases, such as Microsoft SQL Server, is that of transactional integrity. Any transaction must be completely atomic (that is, functionally indivisible) in that all changes must be either applied or not applied, even in the event of a system failure. In a user-defined transaction, all statements bracketed by the **BEGIN TRANSACTION** and **COMMIT TRANSACTION** statements are either applied or not applied. In an implicit transaction, each single SQL statement is considered an atomic unit.

This capability enables SQL Server to experience a power failure, operating system crash, and so forth when in production and after restarting, thus automatically recovering the database to a consistent state, with no human interaction required. This contrasts with non-relational systems which often require lengthy manual procedures to inspect the database for consistency problems following a system failure.

The transaction log mechanism is what provides this capability. Since transactional integrity is considered a fundamental, intrinsic characteristic of SQL Server, logging cannot be disabled. Certain utility or maintenance operations, such as fast BCP and **SELECT INTO**, do minimal logging, but even these log extent allocations so that rollback is possible.

The space requirements for logging can be considerable. For example, in most cases the before and after image of each updated data row must be recorded, plus that of any affected index rows. Since a certain fixed amount of transaction record overhead must be recorded for each logged row, the ratio of updated data to log space consumption will vary depending on the row width. For a narrow row, the amount of log space consumed for a particular **UPDATE**, **DELETE** or **INSERT** could be ten times the data space consumed. For wider rows, the amount of log space consumed will be proportionately less. Log space consumption is an unavoidable consequence of providing transactional integrity. The Database Administrator must provide sufficient log space for his or her particular installation.

The amount of log space required can vary depending on many factors and is very difficult to predict accurately beforehand. While general rule-of-thumb figures, such as 15 to 30 percent of the database size, are sometimes mentioned as a starting point for

sizing the log, in actuality this varies widely. Successful SQL Server installations often do some simple empirical tests to roughly assess the log space requirements for their particular data and applications, and then size their log based on this. Attempting to size the log based solely on calculations and without tests is difficult and often inaccurate.

Several difficult-to-predict factors can account for variation in log space consumption. One factor is the query optimizer. For a given SQL data modification statement, the access plan can vary over time depending on statistical distribution of the data. Different access plans can consume different amounts of log space. Another factor is inevitable internal database fragmentation, which can affect the number of page splits performed. There is nothing that can be done or should be done to examine or affect this process, as SQL Server automatically manages data for the user.

An example of a simple test would be to run `DBCC CHECKTABLE(syslogs)`, which returns the number of 2048-byte data pages in the log, both before and after executing a representative sample of your data modification queries. This can give an approximate idea of the log space requirement for these types of queries. It is usually best to err on the side of excess when providing either log or data disk space for relational databases such as SQL Server.

For SQL Server 7.0 and 2000 class servers, the transaction log has the capability to expand as needed. The amount of growth can be governed by the user or allowed to utilize all available disk capacity. A log file is composed of a number of Virtual Log files. The number and size of these virtual log files are determined by SQL Server and cannot be configured. When a database is first created, each physical log file has a minimum of 2 Virtual Log files. Sometimes the database administrator will enable the "truncate log on checkpoint" option of a database in an effort to avoid log space exhaustion. The intent of this option is to provide an automatic method of truncating the log, mainly for development or test databases which do not rely on log dumps for backup. This option does not disable logging or transactional integrity. It merely causes the checkpoint handler to attempt a log truncation approximately every 60 seconds. Note that the log will not be truncated when issuing a manual checkpoint command in a database with "truncate log on checkpoint" on. This option is always on for the tempdb database, even though this is not indicated in the status column of the `sp_help` stored procedure output.

Even with the "truncate log on checkpoint" option enabled, a number of factors can cause log space exhaustion. These are listed below:

1. A large atomic transaction, especially a bulk UPDATE, INSERT, or DELETE: Each single SQL statement is considered an atomic unit that must be applied or not applied in its entirety. For this reason, all row alterations must be logged, and the transaction cannot be truncated over its duration. For example, if a large bulk INSERT was issued that had a running time of five minutes, the log consumed by this transaction cannot be truncated for this period. The database administrator must provide sufficient log space for the largest bulk operation expected or must perform the bulk operation in smaller groups.

2. An uncommitted transaction: The log can only be truncated prior to the oldest uncommitted transaction. There are several possible causes of an uncommitted transaction, most of which are application errors. These include:

a. A bulk transaction: As considered above, for the duration of a large bulk transaction the log records generated by it cannot be truncated. However, such a transaction also precludes log truncation of other shorter transactions which do commit over the same period.

For example, say the database administrator has sized the log such that it is sufficient for the largest envisioned bulk transaction. Yet while this transaction runs, other shorter data modification statements may also be consuming log space. This log space cannot be truncated since the large bulk transaction started first and hence becomes the oldest uncommitted transaction. The administrator must be aware of the concurrency and log impact of a large bulk transaction, and size the log appropriately.

b. A poorly-designed application which allows for user input or other lengthy activity within a user-defined transaction. For example, after issuing a BEGIN TRANSACTION, an application might prompt the user for input which could take a long time, depending on user behavior. Until the user responds and the application issues a COMMIT, log truncation will not be possible.

c. An application error in which a transaction is not committed: A common cause of this is incorrect handling of the DB-Library call `dbcancel()` within a user-defined transaction. When a query is canceled with `dbcancel()`, the currently executing SQL statement is aborted and rolled back, but the outer transaction is not. The application must be aware of this and issue the necessary ROLLBACK TRANSACTION or COMMIT TRANSACTION statement to close the transaction. Failure to do so can often result in error 3902:

The commit transaction has no corresponding BEGIN TRANSACTION.

It may be useful for the application to send a SELECT @@TRANCOUNT to determine what transaction nesting level exists. However, the application should not blindly do this and then issue COMMIT/ROLLBACK to achieve @@TRANCOUNT=0. This is because if @@TRANCOUNT is ever different from what the application expects, this indicates the application has lost track of the transaction nesting level, which is an application design error. Issuing COMMIT/ROLLBACK at this point could result in applying or aborting unintended transactions, since the application does not know which transactions resulted in the unintended transaction level. Instead, the programmer should debug the application and any stored procedures involved to determine the cause of the unintended transaction level.

d. A network error which does not inform SQL Server of a broken network connection: If the client workstation hangs, reboots, or shuts down within a user-defined transaction, the network layer should inform SQL Server of this. If the network does not properly do this, from the perspective of SQL Server the client will appear to be present, and the open transaction from that client will be maintained. This is a network problem and must

be pursued as such. As a workaround, the administrator may be able to determine though using `sp_who`, `sp_lock`, or a network utility which client session still exists and manually kill it.

e. Transaction not committed due to blocking: In a multi-user environment it is possible for an open transaction to become blocked on locks held by another process. In this case, the transaction will nevertheless remain open, preventing log truncation. To detect this, the programmer or database administrator will need to use `sp_who`, `sp_lock`, or other tools to analyze the concurrency environment. In most cases blocking problems can be reduced or eliminated through proper query, index, and database design.

f. Failed attempt to cancel a data modification query: If the application issues a `dbcancel()` and the query is not canceled due to either a network or SQL problem, the query will continue to run and the transaction will remain open. If you suspect a problem here, use `sp_who` to see if the query is cancelled. If attempting to cancel from a TCP/IP sockets client, try the test from a named pipes client, or run the client application on the server computer using local pipes. This will help discern whether a network or SQL problem is preventing the cancel.

3. Checkpoint handler truncation bandwidth exceeded: Although the log is truncated every 60 seconds, the rate at which this truncation takes place is finite. This scenario is uncommon and the other possible causes of log overflow should be considered and ruled out first before inspecting this possibility. However, it is possible to exceed the maximum truncation rate if many clients are simultaneously issuing large updates. This is similar to a funnel which can only drain fluid at a certain rate, and can be overfilled even while draining. In this scenario the application can be restructured to reduce the number of rows being updated, which should always be a primary design goal for any relational database anyway.

If this is not feasible, the system can be reconfigured for increased disk I/O bandwidth through striping, additional controllers, and so forth. It is common in this case to see the checkpoint handler process spend increasing amounts of time in the `DUMP TRANSACTION` state, as it attempts to keep up with log truncation. Once the truncation threshold is exceeded (see below) you may not see the checkpoint handler ever attempt truncation in that database until the log is cleared.

4. Truncation threshold exceeded: The checkpoint handler essentially does a `DUMP TRANSACTION WITH TRUNCATE_ONLY`. Just as if this was issued manually, it will not always succeed if the log is already full to a certain point. For example, a burst of update activity could fill the log to 95% between visits by the checkpoint handler. When the checkpoint handler attempts truncation, while the log is not completely full, it may be too full to allow truncation. This is because the truncation of the log must itself be logged. The only solution in this case is to use `DUMP TRANSACTION WITH NO_LOG` to manually truncate the log. Using the `NO_LOG` option is not recommended except when absolutely necessary, as it is a non-logged operation during which system failure could introduce database errors.

5. Interactions between any of the above: For example, under normal

conditions in an update-intensive environment, the checkpoint handler truncation rate may keep the log from filling up. If a temporarily open transaction caused by any of the above conditions (such as lock contention) causes the log to fill to say, 50%, there will be much less headroom for handling other update situations, making it much more likely to reach the truncation threshold, at which point automatic truncation will not be possible. Transactions in tempdb are logged like any other database. Since **TRUNCATE LOG ON CHECKPOINT** is on in tempdb, in most cases the log will be truncated and not overflow. However, any of the above circumstances can cause the tempdb log to fill up. Tempdb is usually configured for mixed log and data (sysusages.segmap=7) so data and log operations will contend for the same available space. Certain Transact-SQL constructs such as **GROUP BY**, **ORDER BY DESC**, and so forth, will automatically require tempdb for work space. This will also cause an implicit **BEGIN TRANSACTION** record in tempdb for the work space. This tempdb transaction will continue for the duration of the transaction in the user db, which can defer tempdb log truncation for this period. If the transaction in the user db is halted for any reason, including a blocking lock, or the application not processing dbnextrow() to completion, the transaction in tempdb will likewise be left open, preventing tempdb log truncation. The programmer must debug the application and/or resolve the concurrency issues which cause this.

6. Truncation of the transaction log in SQL Server 7.0 and 2000 class servers is accomplished by truncating Virtual Log Files. If any portion of the active log is resident on a given VLF, that Virtual Log File cannot be truncated. If the active log is resident on all Virtual Log Files, the log cannot be truncated. If autogrowth is enabled and there is space on the volume where the transaction log resides and the maximum file size has not been reached, the transaction log grows by the amount specified in the log file properties.

The following discusses log truncation behavior at SQL startup based on whether **TRUNCATE LOG ON CHECKPOINT** is set.

- If **TRUNCATE LOG ON CHECKPOINT** is set and the log is found to be full at startup time, it will be automatically dumped with no_log.
- **TRUNCATE LOG ON CHECKPOINT** is now the default in master because its log cannot be put on a separate device, so it can never be loaded. The only viable option is to discard the log when it gets full.
- If **TRUNCATE LOG ON CHECKPOINT** is not set, and the log is found to be full at startup time, recovery completes, but the final checkpoint is not written. An administrator can get into the database and dump the log with no_truncate to save the data, then dump with no_log to purge it (or just purge it).

Considerations for Autogrow and Autoshrink Configuration

The default **autogrow** and **autoshrink** settings will work for you with no tuning on many SQL Server systems. However, there are environments where you do not have to turn the settings on or where you may have to adjust the **autogrow** and **autoshrink** parameters. This article gives you some background information to guide you when you select the settings for your environment.

MORE INFORMATION

Here are some things to consider if you decide to tune your **autogrow** and **autoshrink** parameters.

How do I configure the settings?

- You can configure the **autogrow** and **autoshrink** settings with:
 - An ALTER DATABASE statement (SQL Server 2000 only).
 - The **sp_dboption** stored procedure.
 - SQL Enterprise Manager.

You can also configure the **autogrow** option when you create a database.

You can view the current settings through the database properties in SQL Enterprise Manager (SEM) or you can run this Transact-SQL command:

```
sp_helpdb [ [ @dbname= ] 'name' ]
```

- Keep in mind that the **autogrow** settings are per file. Therefore, you have to set them in at least two places for each database (one for the primary data file and one for the primary log file). If you have multiple data and/or log files, you must set the options on each file. Depending on your environment, you may end with different settings for each database file.

What are the performance implications?

- If you run a transaction that requires more log space than is available, and you have turned on the **autogrow** option for the transaction log of that database, then the time it takes the transaction to complete will include the time it takes the transaction log to grow by the configured amount. If the growth increment is large or there is some other factor that causes it to take a long time, the query in which you open the transaction might fail because of a timeout error. The same sort of issue can result from an autogrow of the data portion of your database. To change your **autogrow** configuration, see the "ALTER DATABASE" topic in SQL Server Books Online.
- If you run a large transaction that requires the log to grow, other transactions that require a write to the transaction log will also have to wait until the grow operation completes.

- If you combine the **autogrow** and **autoshrink** options, you might create unnecessary overhead. Make sure that the thresholds that trigger the growth and shrink operations will not cause frequent up and down size changes. For example, you may run a transaction that causes the transaction log to grow by 100 MB by the time it commits. Some time after that the **autoshrink** starts and shrinks the transaction log by 100 MB. Then, you run the same transaction and it causes the transaction log to grow by 100 MB again. In that example, you are creating unnecessary overhead and potentially creating fragmentation of the log file, either of which can negatively affect performance.
- Physical fragmentation from changing the size of the data or log files can have a severe affect on your performance. This is true whether you use the automatic settings or whether you manually grow and shrink the files frequently.
- If you grow your database by small increments, or if you grow it and then shrink it, you can end up with disk fragmentation. Disk fragmentation can cause performance issues in some circumstances. A scenario of small growth increments can also reduce the performance on your system.

Best Practices

- For a managed production system, you must consider **autogrow** to be merely a contingency for unexpected growth. Do not manage your data and log growth on a day-to-day basis with **autogrow**.
- You can use alerts or monitoring programs to monitor file sizes and grow files proactively. This helps you avoid fragmentation and permits you to shift these maintenance activities to non-peak hours.
- **AutoShrink** and **autogrow** must be carefully evaluated by a trained Database Administrator (DBA); they must not be left unmanaged.
- Your **autogrow** increment must be large enough to avoid the performance penalties listed in the previous section. The exact value to use in your configuration setting and the choice between a percentage growth and a specific MB size growth depends on many factors in your environment. A general rule of thumb to you can use for testing is to set your **autogrow** setting to about one-eighth the size of the file.
- Turn on the <MAXSIZE> setting for each file to prevent any one file from growing to a point where it uses up all available disk space.
- Keep the size of your transactions as small as possible to prevent unplanned file growth.

Why do I have to worry about disk space if size settings are automatically controlled?

- The **autogrow** setting cannot grow the database size beyond the limits of the available disk space on the drives for which files are defined. Therefore, if you rely on the **autogrow** functionality to size your databases, you must still independently check your available hard disk space. The **autogrow** setting is also limited by the MAXSIZE parameter you select for each file. To reduce the possibility of running out of space, you can monitor the Performance Monitor counter **SQL Server: Databases Object : Data File(s) Size (KB)** and set up an alert for when the database reaches a certain size.
- Unplanned growth of data or log files can take space that other applications expect to be available and might cause those other applications to experience problems.
- The growth increment of your transaction log must be large enough to stay ahead of the needs of your transaction units. Even with **autogrow** turned on, you can receive a message that the transaction log is full, if it cannot grow fast enough to satisfy the needs of your query.
- SQL Server does not constantly test for databases that have hit the configured threshold for **autoshrink**. Instead, it looks at the available databases and finds the first one that is configured to **autoshrink**. It checks that database and shrinks that database if needed. Then, it waits several minutes before checking the next database that is configured for **autoshrink**. In other words, SQL Server does not check all databases at once and shrink them all at once. It will work through the databases in a round robin fashion to stagger the load out over a period of time. Therefore, depending on how many databases on a particular SQL Server instance you have configured to **autoshrink**, it might take several hours from the time the database hits the threshold until it actually shrinks.

Other References

Please check the following sites for more information about tools that analyze MS SQL log files:

www.lumigent.com

www.apexsql.com



| | | |
|-----------------------------|---------------|--|
| Lebanon | Phone | 961-1-399855 |
| | Fax | 961-1-380420 |
| | PO.Box | 166607 – 1100 2140 Ashrafieh Beirut – Lebanon |
| United Arab Emirates | Phone | 971-4-3383318 |
| | Fax | 971-4-3383319 |
| | PO.Box | 35046 Al Romoul Dubai – U.A.E. |
| Saudi Arabia | | |
| | Riyadh | Phone 966-1-4730784 Fax 966-1-4730784 |
| | Jeddah | Phone 966-2-6608824 Fax 966-2-6603835 |
| | | PO.Box 5586 Riyadh 11432 - Kingdom Of Saudi Arabia |
| | Email | sdcg@softwaredesign.com.lb |
| | Website: | http://www.sd-lb.com |