

# Working with tempdb in SQL Server 2005

SQL Server Technical Article

Writers: Wei Xiao, Matt Hink, Mirek Sztajno, Sunil Agarwal

Technical Reviewer: Wei Xiao

Published: June 2006

Applies To: SQL Server 2005

Summary: This white paper describes how SQL Server 2005 uses **tempdb**. Many improvements in SQL Server 2005 optimize **tempdb** usage and make it easier to manage and to troubleshoot. A case study that uses a workload similar to TPC Benchmark H (TPC-H)  shows new ways to manage and troubleshoot **tempdb** resources. This paper also includes items to consider when upgrading to SQL Server 2005 and configuring **tempdb**.

### Copyright

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

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

 2006 Microsoft Corporation. All rights reserved.

Microsoft and Windowsare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

### Table of Contents

[Introduction 1](#_Toc140465049)

[tempdb Space Usage 1](#_Toc140465050)

[Internal objects 2](#_Toc140465051)

[Version store 2](#_Toc140465052)

[User objects 3](#_Toc140465053)

[Improvements in SQL Server 2005 4](#_Toc140465054)

[tempdb Space Requirements 5](#_Toc140465055)

[Space required by queries 6](#_Toc140465056)

[Space required by features that use version store 7](#_Toc140465057)

[Space required by other features 8](#_Toc140465058)

[Space required for tempdb logging 9](#_Toc140465059)

[Monitoring and Troubleshooting 9](#_Toc140465060)

[Space in tempdb 10](#_Toc140465061)

[Monitoring space 10](#_Toc140465062)

[Troubleshooting space 12](#_Toc140465063)

[I/O in tempdb 14](#_Toc140465064)

[Monitoring I/O 14](#_Toc140465065)

[Troubleshooting I/O 15](#_Toc140465066)

[DML operations in tempdb 16](#_Toc140465067)

[Monitoring contention caused by DML operations 16](#_Toc140465068)

[Troubleshooting contention caused by DML operations 17](#_Toc140465069)

[DDL in tempdb 18](#_Toc140465070)

[Monitoring contention in DDL operations 18](#_Toc140465071)

[Troubleshooting contention in DDL operations 19](#_Toc140465072)

[tempdb Configuration and Maintenance 19](#_Toc140465073)

[tempdb size, including the right number of files and file size 19](#_Toc140465074)

[tempdb log size and version store size 20](#_Toc140465075)

[tempdb auto grow setting 20](#_Toc140465076)

[Shrinking files 20](#_Toc140465077)

[Upgrading to SQL Server 2005 21](#_Toc140465078)

[Best practices and recommendations 21](#_Toc140465079)

[Appendix: A Case Study 22](#_Toc140465080)

[Machine configuration 22](#_Toc140465081)

[Workload 22](#_Toc140465082)

[tempdb space usage monitoring 23](#_Toc140465083)

[Performance warehouse schema 23](#_Toc140465084)

[tempdb space usage collection 25](#_Toc140465085)

[tempdb space usage analysis 26](#_Toc140465086)

[Summary 30](#_Toc140465087)

#### Introduction

The SQL Server system database, **tempdb**, has undergone a number of changes in SQL Server 2005. There are new **tempdb** usages and internal optimizations in SQL Server 2005; **tempdb** architecture is mostly unchanged since SQL Server 2000.

The **tempdb** system database is very similar to a user database. The main difference is that data in **tempdb** does not persist after SQL Server shuts down.

Each time SQL Server restarts, **tempdb** is copied from the model database. It inherits certain database configuration options, such as ALLOW\_SNAPSHOT\_ISOLATION, from the model database.

Only one file group in **tempdb** is allowed for data and one file group for logs. You can configure the size of the files. When auto grow is enabled (which is the default), the file grows until the disk volume is full. When the server restarts, the **tempdb** file size is reset to the configured value (the default is 8 MB). Auto grow is temporary for **tempdb** (unlike other types of databases). It is reset when SQL Server restarts.

Users can explicitly create and use tables in **tempdb**. Transactions can be used to modify data in the tables. Transactions can be rolled back. However, there is no need to REDO them because the contents of **tempdb** do not persist across SQL Server restarts. Because the transaction log does not need to be flushed, transactions are committed faster in **tempdb** than in user databases. In a user database, transactions have the ACID attributes: atomicity, concurrency, isolation, and durability. In **tempdb**, transactions lose the durability attribute.

SQL Server uses **tempdb** to store internal objects such as the intermediate results of a query. Most of these internal operations on **tempdb** do not generate log records because there is no need to roll back. So, these operations are faster.

There are some feature restrictions in **tempdb**. In addition, some of the database options cannot be modified for **tempdb**. For a list of these options, see [tempdb Database](http://msdn2.microsoft.com/en-us/library/ms190768.aspx) in SQL Server 2005 Books Online.

Auto shrink is not allowed for **tempdb**. Database shrink and file shrink capabilities are limited. This is because many of the hidden objects that SQL Server stores in **tempdb** cannot be moved by shrink operations.

The following restrictions also apply to **tempdb**:

* The database CHECKSUM option cannot be enabled.
* A database snapshot cannot be created on **tempdb**.
* DBCC CHECKALLOC and DBCC CHECKCATALOG are not supported.
* Only offline checking for DBCC CHECKTABLE is performed. This means that a TAB-S lock is needed. There are internal consistency checks that occur when **tempdb** is in use. If these checks fail, the user connection is broken and the **tempdb** space used by the connection is freed.

#### tempdb Space Usage

The following types of objects can occupy **tempdb** space:

* Internal objects
* Version stores
* User objects

##### Internal objects

Internal objects are created internally by SQL Server. Applications cannot directly insert into or delete rows from internal objects. Internal object metadata is stored in memory. The metadata does not appear in system catalog views such as **sys.all\_objects**. Internal objects are considered to be hidden objects.

Internal objects are used:

* To store intermediate runs for sort.
* To store intermediate results for hash joins and hash aggregates.
* To store XML variables or other large object (LOB) data type variables. The LOB data type includes all of the large object types: **text**, **image**, **ntext**, **varchar(max)**, **varbinary(max)**, and all others.
* By queries that need a spool to store intermediate results.
* By keyset cursors to store the keys.
* By static cursors to store a query result.
* By Service Broker to store messages in transit.
* By INSTEAD OF triggers to store data for internal processing.

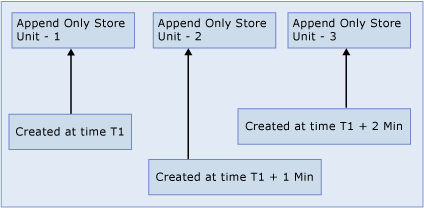
Internal objects are also used by any feature that uses these operations. For example, DBCC CHECK internally uses a query that may need to spool intermediate results. Query notification and event notification use Service Broker, so they need space in **tempdb** as well.

Updates to internal objects do not generate log records. Page allocations on internal objects do not generate log records unless on a sort unit. If the statement fails, these objects are deallocated. Each internal object occupies at least nine pages (one IAM page and eight data pages) in **tempdb**.

##### Version store

Version stores are used to store row versions generated by transactions for features such as snapshot isolation, triggers, MARS (multiple active result sets), and online index build. There are two version stores in **tempdb** for the whole instance of SQL Server. The online index build version store is for row versions from tables that have online index build operations on them. The common version store is for row versions from all other tables in all databases. For more information, see [Row Versioning Resource Usage](http://msdn2.microsoft.com/en-us/library/ms175492.aspx) in SQL Server Books Online.

The version store consists of *append-only store units*. The append-only store units are highly optimized for sequential inserts and random look up. They do not appear in catalog views, such as **sys.all\_objects**. Inserts into the version store do not generate log records. Each unit can store many row versions. If there are versions to be stored, a new store unit is created about every minute. Versions generated by a transaction are stored in the same store unit in each of the two version stores for the life of the version store.



In the illustration, there is one append-only store unit. It is created every minute starting with time T1. Any version generating a transaction that started between times T1 and (T1 + 1) will use *Append-Only Store Unit – 1* for storing any row version it generates, independent of its duration. For example, *Tran1* starts at time *T1* and updates a different row every 30 seconds until it commits at time *T1 + 20*. In this case, all 40 row versions are stored in *Append-Only Store Unit – 1.* The same is true for other transactions that start under different *Append-Only Store Units* as well.

Note that the size of a specific append-only store unit depends on the number of versions that are generated by the transactions that are associated with it. This is reflected in the *Append-Only Store Units* of different sizes in the illustration. However, a query that is running under snapshot isolation may read row versions from any *Append-Only Store Unit* that exists at the time the transaction under snapshot isolation starts until the time it is completed. The same is true for queries that are running under read committed snapshot as well. A background thread periodically deallocates whole append-only store units*,* not individual row versions, that are not needed. This deallocation is done strictly in the order of allocation; therefore, *Append-Only Store Unit-2* can only be deallocated after *Append-Only Store Unit – 1.*

For snapshot isolation and read committed snapshot isolation, the versions are generated by DML operations in the database when the respective database options are enabled. For AFTER triggers, versions are generated for all the update operations by the transaction that fired the trigger during the INSERT, DELETE, or UPDATE statement in any database, independent of database options. INSTEAD OF triggers do not generate versions. For MARS, versions are generated by the UPDATE or DELETE statement when there is a pending SELECT on the same transaction and the same connection. Similarly, building an online index generates versions of the rows while the online index is being built.

##### User objects

User objects appear in catalog views such as **sys.all\_objects**. **Sp\_spaceused** can show the size occupied by these objects. User objects include both user-defined tables and indexes, and system catalog tables and indexes. These two types of tables have exactly the same data structure on disk. User-defined tables can be manipulated by Transact-SQL. System catalog tables cannot be directly modified by Transact-SQL. They are visible by using system catalog views, and they are modified by using DDL statements to ensure logical consistency. Operations on user objects in **tempdb** are mostly logged. Bulk copy program (BCP), bulk insert, SELECT INTO, and index rebuild operations are bulk logged. This is exactly the same as in other databases with the recovery model set to simple.

User-defined tables include the global temporary tables such as ##t, and local temporary tables such as #t. Both of these objects are session scoped but a global temporary table lives until all sessions that are using it expire or terminate. Local temporary tables, on the other hand, are destroyed when the scope (for example, stored procedure or session) they were created in expires or terminates. Local temporary tables also include table variables such as @t, the return value in table valued functions, and the mapping index for online clustered index build with the SORT\_IN\_TEMPDB option.

#### Improvements in SQL Server 2005

Following are some important improvements in **tempdb** in SQL Server 2005.

* The **tempdb** logging optimization avoids logging the “after value” in certain log records in **tempdb**. When an UPDATE operation is performed without this optimization, the before and after values of the data are recorded in the log file. For example, if someone updates a data item from ABC to XYZ, ABC is the before value and XYZ is the after value, which does not need to be logged. For INSERT, there is only the after value (the value being inserted), which does not need to be logged. For DELETE, there is only the before value. This optimization primarily benefits INSERT and UPDATE operations on Heap and LOB data. This optimization can significantly reduce the size of the **tempdb** log as well as reduce the amount of I/O traffic on the **tempdb** log device.
* Instant data file initialization works by not zeroing out the NTFS file when the file is created or when the size of the file is increased. This minimizes overhead significantly when **tempdb** needs to auto grow. Without this, auto grow could take a long time and lead to application timeout. This optimization requires Microsoft® Windows XP or Windows 2003, and it requires special volume permission on the SQL Server service account. For more information, see Instant File Initialization in SQL Server Books Online.
* There is less use of the UP type page latch when allocating pages and extents in **tempdb**. In addition, allocating the first eight pages of a table is more efficient. As a result there should be less contention on the PFS (Page Free Space), SGAM (Secondary Global Allocation Map), and GAM (Global Allocation Map) pages when there are a high number of users using **tempdb** at the same time.
* Proportional fill has been optimized to reduce UP latch contention. Proportional fill means that, when there are multiple data files in **tempdb**, each file is filled in proportion to the free space that is available in the file so that all of the files fill up at about the same time. This was accomplished by removing a latch that was taken during proportional fill. In SQL Server 2000, this latch could become a contention point when there were a high number of users using **tempdb** at the same time. Proportional fill has been optimized to eliminate this latch. With this change in SQL Server 2005, applications will experience less UP latch contention in **tempdb**.
* There is now deferred drop in **tempdb**. This means, for example, that when a large temporary table is dropped by an application, it is handled by a background task and the application does not have to wait. This results in faster response time to applications.
* Worktable caching is improved. When a query execution plan is cached, the work tables needed by the plan are not dropped across multiple executions of the plan but merely truncated. In addition, the first nine pages for the work table are kept. In SQL Server 2000, the work tables used during query plan execution are dropped. Because the work table is cached, the next execution of the query is faster. When the system is low on memory, the execution plan may be removed from the cache and the associated work tables dropped as well. Both SQL Server 2000 and SQL Server 2005 use a small global pool of pre-allocated pages and extents that make the initial creation of work tables faster.
* SQL Server 2005 caches temporary objects. When table-valued functions, table variables, or local temporary tables are used in a stored procedure, function, or trigger, the frequent drop and create of these temporary objects can be time consuming. This can cause contentions on **tempdb** system catalog tables and allocation pages. In SQL Server 2005, these are cached. That means that dropping and creating temporary objects is very fast. When SQL Server drops a temporary object, it does not remove the catalog entry for the object. If a temporary object is smaller than 8 MB, then one data page and one IAM page are also cached so that there is no need to allocate them when re-creating the objects. If a temporary object is larger than 8 MB, defer drop is used. When **tempdb** is low on space, SQL Server frees up the cached temporary objects. You can drop the associated stored procedure(s) or free the procedure cache to get rid of these temporary tables.
* SQL Server 2005 caches the temporary table that is created by using a CREATE TABLE or SELECT INTO statement. In some cases, temporary tables are not cached, such as when there is an explicit DDL on a temporary table after it is created, or if there is a named constraint on the temporary table. Temporary tables are also not cached if they are part of dynamic SQL or ad-hoc batch. The following example shows the creation of a temporary table using dynamic SQL. This table will not be cached.

*sp\_executeSQL N'create table #t(a int)'*

#### tempdb Space Requirements

It is difficult to estimate the **tempdb** space requirement for an application. This section describes the general methodology for estimating the space requirement. These methods are not precise. It requires experience and experiment to gain a satisfactory result. We recommend that you always have a safety factor of about 20% more space. Space estimates must also allow for the future growth of data in the application.

To understand the space requirement of **tempdb** data files, first look at which features of SQL Server 2005 use **tempdb**. The **tempdb** space requirement is different for each feature. The following features use **tempdb**:

* Query
* Triggers
* Snapshot isolation and read committed snapshot (RCSI)
* MARS
* Online index creation
* Temporary tables, table variables, and table-valued functions
* DBCC CHECK
* LOB parameters
* Cursors
* Service Broker and event notification
* XML and LOB variable
* Query notifications
* Database mail
* Index creation
* User-defined functions

A server instance may use some or all of these features. For some servers, one or two features might dominate **tempdb** space usage. In this case, we concentrate most of the capacity planning effort on these few features. For example, you may find out that for one server, a query requires a peak of 50 GB of **tempdb** space, and RCSI requires a peak of 20 GB of **tempdb** space, and the remaining features require only 1 GB of peak space. It would be sufficient to allocate 85 GB (71GB + 20% overhead) of **tempdb** space for this server.

##### Space required by queries

Applications usually send many queries to SQL Server. If queries are sent to the server serially, the peak **tempdb** space requirement is determined by the query that requires most of the **tempdb** space. If queries are executed at about the same time, the total **tempdb** space requirement is the sum of the requirement of all queries.

SQL Server has a cost-based query optimizer. It chooses a query plan that has the lowest execution cost but it ignores the implications of this plan on **tempdb**. As a result, the **tempdb** space requirement for a query may vary, depending on the plan. In some cases, the best plan according to the query optimizer may require more **tempdb** space than a less efficient plan would. Also, when an application is upgraded from SQL Server 2000, a plan change may require more space in **tempdb**.

For capacity planning, prepare for the worst-case scenario. Otherwise a query may fail due to a space error in **tempdb**. In some cases, auto grow can be used to expand a **tempdb** file until there is no space remaining in the disk volume.

To estimate the maximum **tempdb** space required for a query, look at the query plan. There are several ways to view a query plan in SQL Server. SQL Server Management Studio can show the query plan that is in use, or estimate a query plan in graphical format. For more information, see [Analyzing a Query](http://msdn2.microsoft.com/en-us/library/ms191227.aspx) in SQL Server Books Online.

The common operators that require **tempdb** space are listed in the following table. Each of these operators consumes some number of rows and outputs some number of rows.

Table 1. Query operators that use tempdb

|  |  |
| --- | --- |
| Sort (including distinct sort) | The sort operator needs **tempdb** space to sort the full rowset (the rowset that goes through the sort operator). |
| Hash match | This operator has two inputs—one to build the hash table and other to probe it. Depending on its size, a hash table might be in **tempdb**. Look at the number of rows and row size returned by the first input operator. |
| Spool (including table spool, nonclustered index spool) | This operator requires that the full input rowset be stored in **tempdb**. |

To estimate the space required by each operator, look at the number of rows and the row size reported by the operator. To calculate the space required, multiply the actual (or estimated) number of rows by the estimated row size. Be aware that the estimated number of rows and estimated row size can be incorrect by a very large margin, due to incorrect statistics and other issues. Adjust your estimates based on your knowledge of the data.

##### Space required by features that use version store

The *version store* is a collection of hidden objects in **tempdb** that support the following features:

* Snapshot isolation
* Read committed snapshot isolation (RCSI)
* Online index build
* Triggers
* MARS

There are two version stores. One of these stores row versions that are generated when an online index is built. The second stores row versions that are generated for the other operations mentioned in the previous bulleted list. New versions are inserted into the version store when there are DML operations in a database. DML operations might be UPDATE, DELETE, and sometimes INSERT (if inserting a row with the same unique key as the row that was deleted). Old versions are removed when they are no longer needed. The size of the version store depends on two factors: how long the versions are needed and how many versions are generated.

One of the two version stores is for tables that are under online index build operations. The versions in this store are needed as long as an online index build transaction is active. For a very large index, these transactions could be active for hours or even days. Fortunately, the versions in this store are only from the tables with active online index build operations, so this could be a very small subset of all the versions that could be generated in a large system. For example, you are building a new nonclustered index online on a big table. You estimate that this will take about 12 hours. During this time, you estimate that about 600 data records in this table will be changed per hour by the concurrent DML. The version store keeps the entire record, so you assume that the average record size of the table is 1 KB. The space required in the version store is 12\*600\*1KB = 7200KB.

The other version store is for versions generated on all tables except those that are under active online index build. The versions in this store are needed as long as there is an active transaction that uses one of these features: snapshot isolation, read committed snapshot isolation (RCSI), triggers, or MARS.

For example, if the longest running transaction in your SQL Server instance is a snapshot isolation transaction that is doing a complex report that takes five hours, then the versions are needed in the version store for five hours. Different features use version stores in different ways. Keep this in mind when estimating the number of version records generated per minute.

For snapshot isolation and RCSI, as long as the database option is set, any update in the database is versioned. For example, the maximum update rate is 1,000 records per minute in all the databases with the options set, and the average record size is 1 KB. The version store space required for this system with five hours of maximum transaction time is approximately 5\*60\*1000\*1KB=300MB. If a snapshot isolation transaction starts concurrently before the first transaction commits, the versions are kept until the snapshot isolation is complete. So the version store must be kept for more than five hours. A general method for estimating the size of version store is to multiply the size of the version store that was generated during the execution of the longest transaction by two. When the database is enabled for snapshot isolation or RCSI, only the first update to a particular record in a transaction generates a version.

For triggers, the original triggering DML plus all the updates that are done inside the trigger body generate a version.

For MARS, all updates generate versions when there is a concurrent active rowset that returns results, such as the results that are returned by a SELECT statement.

##### Space required by other features

Temporary tables, table variables, and table-valued functions all use space in **tempdb**. The application has explicit control over how much data goes into these objects. The space occupied by these objects should be estimated in the same way as user tables.

LOB variables (including parameters) and XML variables all consume space in **tempdb**. This is new in SQL Server 2005. In SQL Server 2000, these variables were only memory-resident. The intermediate result of a LOB or XML expression can also consume space in **tempdb**. To estimate the space requirement, look at the programming logic and try to find a reasonable upper limit of the resource requirement.

DBCC CHECK internally starts a query to verify the consistency of data. The execution plan of this query cannot be displayed. However, there is an option to estimate the space requirement for this command. If you suspect that there may not be enough **tempdb** space, estimate the space requirement before you run the command. Otherwise the application might fail because DBCC CHECK is competing with the application for **tempdb** space.

Service Broker uses about 1 MB per dialog. Unless the application starts a lot of dialogs without closing them, space usage should not be an issue. Also keep in mind that other features, such as event notification, use Service Broker.

Index build has an option to sort in **tempdb**. The sort requires about the same amount of **tempdb** space as does the index that is being built. For online index build, we may also need a mapping index. To calculate the size of this index, multiply the average key size by the number of rows in the index.

Two kinds of server-side cursors require **tempdb** space: static and keyset. For static cursors, space is required for the whole result. For keyset cursors, the space required is the average key size multiplied by the number of rows in the result set.

##### Space required for tempdb logging

Most operations in **tempdb** are not logged. Operations on temporary tables, table variables, table-valued functions, and user-defined tables are logged. These are called *user objects*. Sort operations in **tempdb** also require logging for activities related to page allocation. The log size requirement depends on two factors: how long it is necessary to keep the log and how many log records are generated during this time.

Estimating the log space for **tempdb** is similar to estimating the log file size for other databases. However, when you estimate the **tempdb** log size, you pay attention only to operations that generate log records. For example, the longest running transaction in **tempdb** is an index build operation that uses sort. This transaction ran for five hours. An estimated 1,000 temporary table records were fully overwritten per minute by updates. The average record size is 1 KB. The calculation for the log space required is 5\*60\*1000\*1 KB = 300 MB.

Since log truncation can become a contention point in a heavily loaded system with a small log file, make the **tempdb** log file large enough to avoid truncating logs frequently. For example, if 50 MB of the log file can be generated per second in the system, the log size should be at least 500 MB so that a log truncation is performed about every 10 seconds.

#### Monitoring and Troubleshooting

The **tempdb** shared resource must be managed well for the smooth functioning of SQL Server. A rogue query or a poorly designed application can potentially consume all **tempdb** resources, thereby rendering a SQL Server instance unusable.

Administrators work with the following types of resources in **tempdb**:

* Space. This refers to both used and unused space in data and log files.
* I/O bottlenecks.
* Contention in DML operations. This relates to page and extent allocations and deallocations for all type of objects in **tempdb**.
* Contention in DDL operations. This relates to the manipulation of metadata when user objects in **tempdb** are created or dropped. Note that internal objects are not reflected in metadata tables

One of challenges in managing **tempdb** is that there is no way to partition its resources based on user databases, applications, or user sessions. You cannot even partition resources based on the category of objects (such as version stores) that are stored. It seems useful to partition **tempdb** resources to isolate applications from one another but then the burden is on the applications or on the administrators to partition these resources in such a way so as to minimize waste. For example, if you put an upper bound on the **tempdb** resources that can be consumed by an application, you may be forced to abort one or more transactions even if those resources were not being used by any other applications or queries.

In SQL Server, these resources are managed as a global pool. SQL Server 2005 provides a rich set of tools for monitoring resources in **tempdb** so that you can isolate applications or queries that are consuming significant resources in **tempdb** with relative ease and take appropriate corrective action.

This section describes which resources you need to monitor and the tools available to you to accomplish this. It also covers how to troubleshoot problems related to specific resources. Active monitoring is useful to spot the problems early so that you can take corrective action before problems become catastrophic. There are significant improvements to the ways you can monitor **tempdb** resources in SQL Server 2005.

##### Space in tempdb

Most queries and DML statements allocate space in **tempdb** either implicitly (for example, by using work tables) or explicitly (for example, by using user-defined temporary tables). If there is no space left in **tempdb**, the server can become unusable. You might run out of space in **tempdb** when it reaches its maximum size limit or if there is no space remaining in the physical disk to store **tempdb** data and log files. The only course of action at this stage is to kill one or more sessions and/or transactions to free up space in **tempdb**.

Space management in **tempdb** is even more critical in SQL Server 2005 as it has new features (such as row versioning, MARS, triggers, and online index) that use **tempdb** heavily. Since **tempdb** is a global resource, unless the **tempdb** space used by the set of applications that are running on the server is very well understood, there is very little a system administrator can do to control **tempdb** space usage except to monitor it and act appropriately when the available space in **tempdb** falls below critical limit.

###### Monitoring space

This section describes performance counters and dynamic management views (DMVs) that you can use to monitor space usage in **tempdb**.

Performance Counters

Database: Log File(s) Size(KB)

Returns the cumulative size of all the log files in the database. This size can grow if you have not set a maximum size for the log in **tempdb**.

Database: Log File(s) Used (KB)

Returns the cumulative used size of all log files in the database. A large active portion of the log in **tempdb** can be a warning sign that a long transaction that is preventing log cleanup.

Free Space in tempdb (KB) (SQL Server 2005)

Tracks free space in **tempdb** in kilobytes. Administrators use this counter to determine if **tempdb** is running low on free space so they can take appropriate corrective action. This counter accounts for space allocated by all three types of objects in **tempdb**.

Version Store Size (KB) (SQL Server 2005)

Monitors the size in KB in both version stores. If a version store is not shrinking, it implies that a long-running transaction is preventing version store cleanup.

Version Generation Rate (KB/s) (SQL Server 2005)

Monitors the version generation rate in kilobytes per second (KBps) in both version stores.

Version Cleanup Rate (KB/s) (SQL Server 2005)

Monitors the version cleanup rate in KBps in all version stores. If the version cleanup rate is lower than the version generation rate, the version store will use more and more space in **tempdb**. However, if the version cleanup rate is 0 but the version generation rate is not, there is probably a long-running transaction that is preventing the version store cleanup.

Dynamic Management Views

SQL Server 2005 provides a rich set of DMVs to track **tempdb** space usage. You can track **tempdb** space usage at the instance level, the session level, or by individual task (a task is also known as a *batch*) by using the following DMVs.

Sys.dm\_db\_file\_space\_usage

This DMV returns space allocation information for the files associated with **tempdb**. The allocation information is grouped by object category (user, internal, and version store) for the instance. The following code is an example.

SELECT

SUM (user\_object\_reserved\_page\_count)\*8 as usr\_obj\_kb,

SUM (internal\_object\_reserved\_page\_count)\*8 as internal\_obj\_kb,

SUM (version\_store\_reserved\_page\_count)\*8 as version\_store\_kb,

SUM (unallocated\_extent\_page\_count)\*8 as freespace\_kb,

SUM (mixed\_extent\_page\_count)\*8 as mixedextent\_kb

FROM sys.dm\_db\_file\_space\_usage

The following sample output was generated when a query was executed with a hash-join that created a work table.

usr\_obj\_kb internal\_obj\_kb version\_store\_kb freespace\_kb mixedextent\_kb

8192 4608 0 3840 1024

You can use the output to learn how **tempdb** space is being used. The following points will help you analyze the output.

* A higher % allocation for user objects implies that objects that are created by applications (for example, global and local temporary tables and variables) are the major consumers of **tempdb**. This is not necessarily a cause of concern.
* A higher % allocation for internal objects implies that the query plans make heavy use of **tempdb**. This is not necessarily a problem, but you may want to look at the query plans to see if alternate query plans can be generated by creating indexes or by re-formulating the queries so as to minimize **tempdb** space usage.
* A higher % allocation for the version store implies that version store cleanup cannot keep pace with version generation. See if a long-running transaction is preventing version store cleanup. Or, a high transaction throughput might be generating a large number of versions per minute. The background task cleans up versions every minute.

Sys.dm\_db\_session\_file\_usage

This DMV tracks the historical allocation/deallocation of pages in **tempdb** for the active sessions. A session is established when a user connects to the database. The session is active until the connection is terminated. During the course of the session, the user submits one or more batches. This DMV tracks the **tempdb** space usage only by the completed batches. The following code example shows the top five sessions that have allocated a maximum space for user objects and internal objects in **tempdb**. This represents the batches that have already completed, but the code lists sessions with heavy **tempdb** space use. You could look at these first if you want to minimize **tempdb** consumption. Note that this **tempdb** space usage does not take into account the impact of this session on the version store space.

SELECT top 5 \*

FROM sys.dm\_db\_session\_space\_usage

ORDER BY (user\_objects\_alloc\_page\_count +

internal\_objects\_alloc\_page\_count) DESC

Sys.dm\_db\_task\_space\_usage

This DMV tracks the allocation/deallocation of **tempdb** pages by the currently executing tasks (also called batches). This is extremely useful when you are running out of space in **tempdb**. Using this DMV, you can identify tasks with heavy **tempdb** space use and optionally kill them. You can then analyze why these tasks require heavy **tempdb** space usage and take corrective action. You can join this DMV with other DMVs to identify the SQL statement and its corresponding query plan for deeper analysis. The following query shows the top five tasks that are currently executing tasks and consuming the most **tempdb** space. The **tempdb** space usage returned does not allow for the impact on space consumed by the version store.

SELECT top 5 \*

FROM sys.dm\_db\_task\_space\_usage

ORDER BY (user\_objects\_alloc\_page\_count +

internal\_objects\_alloc\_page\_count) DESC

###### Troubleshooting space

The first thing to do to troubleshoot space problems in **tempdb** is to configure **tempdb** for your workload. Monitoring **tempdb** size over time will help you determine the optimal size. Like any other database, **tempdb** grows incrementally (either by a fixed size or a percentage) based on the workload. The maximum size a file can grow to is determined by the upper bound of the MAXSIZE of each file or by the available space in the physical device. For example, if the optimal size of your **tempdb** is 1 GB, then you should configure **tempdb** with this starting size to minimize fragmentation. After that, continue to monitor **tempdb** resources and, when needed, take appropriate corrective action.

Tools for troubleshooting

Before SQL Server 2005, troubleshooting **tempdb** space was difficult because:

* There was no easy way (short of dumping the space management structures) to estimate the **tempdb** space used by internal objects. This made it hard to estimate how much free space was available in **tempdb**.
* There was no visibility of **tempdb** usage at a session or task level. If **tempdb** was running out of space, it was difficult to determine which query or session was consuming the majority of resources. That made it easy to make the mistake of killing the wrong session or task.

The following tools in SQL Server 2005 for monitoring space usage in **tempdb** make it easier to troubleshoot problems.

* Use the new DMVs to analyze which Transact-SQL statements are the top consumers of **tempdb** space as described in [Monitoring space](#_Monitoring_space) in this paper. For example you can use the following query that joins the **sys.dm\_db\_task\_space\_usage** and **sys.dm\_exec\_requests** DMVs to find the currently active requests, their associated TSQL statement, and the corresponding query plan that is allocating most space resources in **tempdb**. You may be able to reduce **tempdb** space usage by rewriting the queries and/or the stored procedures, or by creating useful indexes.

SELECT t1.session\_id, t1.request\_id, t1.task\_alloc,

  t1.task\_dealloc, t2.sql\_handle, t2.statement\_start\_offset,

  t2.statement\_end\_offset, t2.plan\_handle

FROM (Select session\_id, request\_id,

    SUM(internal\_objects\_alloc\_page\_count) AS task\_alloc,

    SUM (internal\_objects\_dealloc\_page\_count) AS task\_dealloc

  FROM sys.dm\_db\_task\_space\_usage

  GROUP BY session\_id, request\_id) AS t1,

  sys.dm\_exec\_requests AS t2

WHERE t1.session\_id = t2.session\_id

  AND (t1.request\_id = t2.request\_id)

ORDER BY t1.task\_alloc DESC

Note that if a query is executing in parallel, each parallel thread runs under the same <session-id, request-id> pair.

* You can actively monitor free space in **tempdb** by using the perfmon **free space in tempdb (KB)** counter*.* If space in **tempdb** is critically low, query the **sys.dm\_db\_task\_space\_usage** DMV to find out which tasks are consuming the most space in **tempdb**. You can kill such tasks, where appropriate, to free space.
* If the version store is not shrinking, it is likely that a long-running transaction is preventing version store cleanup. The following query returns the five transactions that have been running the longest and that depend on the versions in the version store.

SELECT top 5 transaction\_id, transaction\_sequence\_num,

elapsed\_time\_seconds

FROM sys.dm\_tran\_active\_snapshot\_database\_transactions

ORDER BY elapsed\_time\_seconds DESC

* **tempdb** can only be configured in the simple recovery model. Typically, the transaction log is cleared with the implicit or the explicit checkpoints. An active long-running transaction can prevent transaction log cleanup and can potentially use up all available log space. To identify a long-running transaction, query the **sys.dm\_tran\_active\_transactions** DMV to find the longest running transaction and, if appropriate, kill it.

##### I/O in tempdb

Like user databases, the data pages in **tempdb** incur physical I/Os under memory pressure in SQL Server. The lazy writer flushes dirty data pages in the background. A physical I/O is incurred when the requested page is not found in the buffer pool. However, the log pages in **tempdb** do not need to be flushed when a transaction is committed. This is because **tempdb** is re-created when SQL Server restarts. The log pages are only flushed under memory pressure. A bottleneck in **tempdb** I/Os can impact the overall throughput of your SQL Server.

###### Monitoring I/O

The first step in solving performance issues is to identify the resources that are experiencing bottlenecks. For example, if CPU is 100% used, this indicates a bottleneck in CPU resources. Similarly, an I/O bottleneck is indicated if I/O requests are queuing up. You can identify I/O bottlenecks by monitoring the following perfmon counters for physical devices associated with **tempdb**.

**PhysicalDisk Object: Avg. Disk Queue Length:** The average number of physical read and write requests that were queued on the selected physical disk during the sampling period. If the I/O system is overloaded, more read/write operations will be waiting. If the disk queue length exceeds a specified value too frequently during peak usage of SQL Server, there might be an I/O bottleneck.

**Avg. Disk Sec/Read**: The average time, in seconds, of a read of data from the disk. Use the following to analyze numbers in the output.

* Less than 10 milliseconds (ms) = very good
* Between 10-20 ms = okay
* Between 20-50 ms = slow, needs attention
* Greater than 50 ms = serious IO bottleneck

**Avg. Disk Sec/Write**: The average time, in seconds, of a write of data to the disk. See the guidelines for the previous item, **Avg. Disk Sec/Read**.

**Physical Disk: %Disk Time**: The percentage of elapsed time that the selected disk drive was busy servicing read or write requests. A general guideline is that if this value > 50 percent, there is an I/O bottleneck.

**Avg. Disk Reads/Sec**: The rate of read operations on the disk. Make sure that this number is less than 85 percent of disk capacity. Disk access time increases exponentially beyond 85 percent capacity.

**Avg. Disk Writes/Sec**: The rate of write operations on the disk. Make sure that this number is less than 85 percent of the disk capacity. Disk access time increases exponentially beyond 85 percent capacity.

**Database: Log Bytes Flushed/sec**: The total number of log bytes flushed. A large value indicates heavy log activity in **tempdb**.

**Database:Log Flush Waits/sec**: The number of commits that are waiting on log flush. Although transactions do not wait for the log to be flushed in **tempdb**, a high number in this performance counter indicates and I/O bottleneck in the disk(s) associated with the log.

###### Troubleshooting I/O

If you determine that a query or application slowdown is caused by an I/O bottleneck in **tempdb**, troubleshoot as follows:

* Identify queries that consume large amounts of space in **tempdb** and see if alternate query plans can be used to minimize the amount of space required by the query.
* See if you have a memory bottleneck that is manifesting itself into I/O problem.
* See if you have a slow I/O subsystem.

Use optimal execution plans

Examine execution plans and see which plans result in more I/O. It is possible that by choosing a better plan (for example, by forcing an index usage for a better query plan), that you can minimize I/O. If there are missing indexes, run Database Engine Tuning Advisor to find the missing indexes. In SQL Server 2005, you can use the following DMV to identify and analyze the queries that are generating the most I/Os.

SELECT top 10 (total\_logical\_reads/execution\_count),

  (total\_logical\_writes/execution\_count),

  (total\_physical\_reads/execution\_count),

  Execution\_count, sql\_handle, plan\_handle

FROM sys.dm\_exec\_query\_stats

ORDER BY (total\_logical\_reads + total\_logical\_writes) Desc

To get the text of the query, run the following DMV query.

SELECT text

FROM sys.dm\_exec\_sql\_text (<sql-handle>

You can examine the query plan by using the following DMV query.

SELECT \*

FROM sys.dm\_exec\_query\_plan (<plan\_handle>\

Check memory configuration

Check the memory configuration of SQL Server. If SQL Server is configured with insufficient memory, it incurs more I/O overhead. You can examine following perfmon counters to identify memory pressure:

* Buffer Cache hit ratio
* Page Life Expectancy
* Checkpoint pages/sec
* Lazywrites/sec

Increase I/O bandwidth

Add more physical drives to the current disk arrays. You could also replace your disks with faster drives. This helps to boost both read and write access times. Do not add more drives to the array than your I/O controller can support.

##### DML operations in tempdb

DML operations (INSERT, DELETE, and UPDATE) that manipulate data often lead to the allocation and deallocation of pages in **tempdb**. Excessive page allocation/deallocation activity in **tempdb** can lead to contention in the allocation structures that track allocation information. For example, PFS pages track the free space in the page, GAM pages track the allocated extents, and SGAM pages track allocation in the mixed extents. Any contention in these meta-pages can affect the performance on the workload. This section describes how to detect if a SQL Server slowdown is due to a contention in allocation structures and how to troubleshoot this condition.

###### Monitoring contention caused by DML operations

Any page allocation/deallocation or any space-changing operation (such as INSERT) acquires a latch on PFS, SGAM or GAM pages to update the space tracking information. Since most temporary objects are heaps and are less than 64 KB in size, a large number of such operations can cause excessive waits when acquiring latches. This can significantly degrade application throughput. Following are some tools that you can use to monitor the contention in allocation structures.

Perfmon counters

Use these performance counters to monitor allocation contention in SQL Server.

* **Access Methods::Worktables Created/sec:** The number of work tables created per second. Work tables are temporary objects and are used to store results for query spool, LOB variables, and cursors. Typically, this number is < 200.
* **Access Methods::Workfiles Created/sec:** The number of work files created per second. Work files are similar to work tables but are created strictly by hashing operations. Work files are used to store temporary results for hash joins and hash aggregates.
* **Access Methods: Worktables From Cache Ratio:** The percentage of work tables that were created where the initial two pages of the work table were not allocated but were immediately available from the work table cache. In SQL Server 2000, there is no caching of temporary tables.
* **Temp Tables Creation Rate** **(SQL Server 2005):** The number of temporary table or variables created/sec.
* **Temp Tables For Destruction** **(SQL Server 2005):** The number of temporary tables or variables waiting to be destroyed by cleanup system thread.

Query the system table

If you experience allocation bottlenecks, query the **sysprocesses** table. Any time you see tasks waiting to acquire latches on **tempdb** pages, you can analyze to see if this is due to PFS and SGAM pages. If it is, this implies an allocation contention in **tempdb**.

You can look for PFS and SGAM pages by using **waitresource**. For example ‘2:1:1’ or ‘2:1:3’ refer to the first PFS page and the first SGAM page in file1 of the **tempdb** database (id=2). SGAM pages re-occur after every 511,232 pages and each PFS page occurs after every 8,088 pages. You can use this information to find all other PFS and SGAM pages across all files in **tempdb**.

The following code example queries the **sysprocesses** table for all latch waits on all pages in database id = 2.

SELECT \*

FROM sys.sysprocesses

WHERE lastwaittype like 'PAGE%LATCH\_%' AND waitresource like '2:%'

Query DMVs

In SQL Server 2005, you can accomplish the same thing by using the DMV **sys.dm\_os\_waiting\_tasks** as in the following code example.

SELECT session\_id, wait\_duration\_ms, resource\_description

FROM sys.dm\_os\_waiting\_tasks

WHERE wait\_type like 'PAGE%LATCH\_%' AND resource\_description like '2:%'

Once you know the page number, you can use the following query to find the type of the page and the object it belongs to. If you see the contention in PFS, GAM or SGAM pages, it implies contention in allocation structures.

SELECT P.object\_id, object\_name(P.object\_id) as object\_name,

P.index\_id, BD.page\_type

FROM sys.dm\_os\_buffer\_descriptors BD, sys.allocation\_units A,

sys.partitions P

WHERE BD.allocation\_unit\_id = A.allocation\_unit\_id and

A.container\_id = P.partition\_id

###### Troubleshooting contention caused by DML operations

If you determine that the throughput of your application has degraded because of contention in allocation structures, you can use the following techniques to minimize it.

* Evaluate your application and the query plans to see if you can minimize the creation of work tables and temporary tables. Monitor the perfmon counters as described in [Monitoring contention caused by DML operations](#_Monitoring_contentions_caused_by DM). Then, use SQL Profiler to correlate the values of these counters with the currently running queries. This helps you identify the queries that are causing the contention in allocation structures.
* Divide **tempdb** into multiple data files of equal size. These multiple files don’t necessarily be on different disks/spindles unless you are also encountering I/O bottlenecks as well. The general recommendation is to have one file per CPU because only one thread is active per CPU at one time. SQL Server allocates pages for **tempdb** objects in a round-robin fashion (also referred to as *proportional fill*) so that the latches on PFS and SGAM pages are distributed among multiple files. This is supported both in SQL Server 2000 and SQL Server 2005. There are improvements to the proportional fill algorithm in SQL Server 2005.
* Use TF-1118. Under this trace flag SQL Server allocates full extents to each **tempdb** object, thereby eliminating the contention on SGAM page. This is done at the expense of some waste of disk space in **tempdb**. This trace flag has been available since SQL Server 2000. With improvements in **tempdb** object caching in SQL Server 2005, there should be significantly less contention in allocation structures. If you see contention in SGAM pages, you may want to use this trace flag. Cached **tempdb** objects may not always be available. For example, cached **tempdb** objects are destroyed when the query plan with which they are associated is recompiled or removed from the procedure cache.

##### DDL in tempdb

Contention for DDL operations only occurs for user objects in **tempdb**. Metadata for internal objects, such as work tables, is cached and is not created in the system catalog. However, metadata for user objects, such as local and global temporary tables and table variables, is created in the system catalog. If your workload or application creates and drops a large number of user objects in **tempdb**, contention in system catalog tables might cause throughput to slow down.

###### Monitoring contention in DDL operations

Use these methods to monitor DDL contention in **tempdb**.

Queries and DMVs

You can use queries as described in [Monitoring contention caused by DML operations](#_Monitoring_contention_caused_by DML) to monitor contention in DDL operations. These queries can be used to identify the latch contention in pages. If the page that is incurring a latch contention is a page that belongs to system catalog, it means you have DDL contention.

Perfmon counters

Use these performance counters to monitor DDL contention.

* **Temp Tables Creation Rate** (SQL Server 2005): The number of temporary tables or variables that are created per second. The value depends on the workload. If your workload creates many temporary tables and the temporary table cache cannot be used, you may see high values. In such cases investigate why the temporary table cache cannot be used; this might be by design in the application. For example, if there is an explicit DDL after the temporary table is created, the table cannot be cached. In a perfect scenario, if there were 100% temporary table cache hits, this counter would be 0
* **Temp Tables For Destruction** (SQL Server 2005): The number of temporary tables or variables waiting to be destroyed by the cleanup system thread. The number of temporary tables or variables waiting to be destroyed by the cleanup system thread should be 0. Temporary spikes are expected, but should be rare. Spikes can happen when the temporary table cache or the procedure cache is being cleaned up.

###### Troubleshooting contention in DDL operations

Evaluate your application and query plans and see if you can minimize the creation of temporary tables. To do this, monitor the perfmon counters **Temp Tables Creation Rate** and **Temp Tables For Destruction**. You can also run SQL Profiler to correlate the values of these counters with the currently running queries. This will help you identify the queries that are causing the contention in system catalog. This might occur, for example, if a temporary object is being created inside a loop or a stored procedure.

Verify if temp objects (temp tables and variables) are being cached. SQL2005 caches Temp objects only when the following conditions are satisfied:

* Named constraints are not created.
* Data Definition Language (DDL) statements that affect the table are not run after the temp table has been created, such as the CREATE INDEX or CREATE STATISTICS statements.
* Temp object is not created by using dynamic SQL, such as: sp\_executesql N'create table #t(a int)'.
* Temp object is created inside another object, such as a stored procedure, trigger, and user-defined function; or is the return table of a user-defined, table-valued function.

#### tempdb Configuration and Maintenance

**tempdb** is re-created each time SQL Server starts. The data files are initialized and the **tempdb** size is set to the default, unless it was explicitly set using the ALTER DATABASE command. This section covers the following considerations for configuring and maintaining **tempdb**:

* **tempdb** size, including the right number of files and file size
* **tempdb** log size
* Version store size
* **tempdb** auto grow setting
* Shrinking files

##### tempdb size, including the right number of files and file size

**tempdb** supports only one data filegroup and one log filegroup. By default, the number of files is set to 1. Multiple files can be created for each filegroup. Adding more data files may help to solve potential performance problems that are due to I/O operations. Increasing the number of files helps to avoid a latch contention on allocation pages (manifested as a UP-latch). The recommended solution is to set the number of files to match the number of CPUs that are configured for the instance. This is only a recommendation—the number of files might not be the same as the number of CPUs.

Having too many files increases the cost of file switching, requires more IAM pages, and increases the manageability overhead. The size of each file is configurable. More files can be added, moved, or removed. To better use the allocation mechanism (proportional fill), the size of the files should be equal. Generally, for SQL Server 2005 it is recommended that you create **tempdb** files striped across fast disks.

To estimate the data file size, three types of **tempdb** space containers should be taken into account:

* user objects
* internal objects
* version store

The size allocated for user objects depends on the user application and on the amount of space that is created by the application for user-defined tables, global and local temporary tables, indexes, and table variables. In addition, any clustered index online operation executed by the database administrator (DBA), such as to create or alter a clustered index, generates a special object called a *mapping index* in the user space. To obtain the size of user objects used for **tempdb** see [Monitoring space](#_Monitoring_space) in this white paper.

Since an application cannot directly insert or delete rows from internal objects, the size of internal objects depends primarily on the size of the objects that are created internally by SQL Server. Query operations such as hash joins, hash aggregates, group by, order by, and index sorts (when SORT\_IN\_TEMPDB specified) allocate space as internal objects. The same applies to operations related to Service Broker, DBCC, XML, or LOB variables. To obtain the size of internal objects used for **tempdb**, see [Monitoring space](#_Monitoring_space).

##### tempdb log size and version store size

The **tempdb**, log, and version store size are discussed in [tempdb Space Usage](#_tempdb_Space_Usage).

##### tempdb auto grow setting

**tempdb** does not require high maintenance. By default, **tempdb** is set to auto grow; once the space in the file is full, the file is allowed to grow 10 % of the initial file size. However, relying on auto grow to manage file growth causes all applications or internal operations that use **tempdb** to pause. Instant data file initialization (see [Improvements in SQL Server 2005](#_Improvements_in_SQL_Server 2005)) shortens this pause, improves performance, and allows files to grow much faster as compared to SQL Server 2000. It is important to note that auto grow can lead to data fragmentation and should be used as a last resort.

##### Shrinking files

Shrinking files is not a recommended practice, since these files will probably grow again. In addition, shrink operations cause data fragmentation. Performing shrink operation on **tempdb** has the following limitations:

* Shrink operations do not shrink the version store or internal objects. This means that you will not recover space if the internal object or version store page needs to be moved.
* The DBA might need to restart the server in a single user mode to allow shrinking of the **tempdb**.

For more information, see [How to shrink the tempdb database in SQL Server](http://support.microsoft.com/default.aspx?scid=KB;EN-US;307487) on the Microsoft Developer Network (MSDN).

##### Upgrading to SQL Server 2005

When the database is upgraded from a previous release to SQL Server 2005, you may need to adjust the disk space used by **tempdb**. There are multiple factors that impact the space requirements for **tempdb**. These are:

* New features, such as snapshot isolation, MARS, and online indexes, that use the space in **tempdb**.
* Query plan changes.
* Features that did not use **tempdb** in previous versions, but use **tempdb** in SQL Server 2005, such as triggers that are based on row versioning and use **tempdb** to store inserted/deleted rows.

For more information on upgrade recommendations, see SQL Server Books Online.

##### Best practices and recommendations

Before you upgrade to SQL Server 2005, you may want to follow the steps outlined here to determine the appropriate size of **tempdb**.

Estimate the size of **tempdb**:

* Use your current **tempdb** size setting as a baseline.
* Analyze the existing workload.
* Evaluate required features.
* Adjust the space for projected concurrent activities.

Create a test environment and monitor **tempdb** space:

* Execute individual queries.
* Run selected workloads.
* Perform index maintenance operations.
* Set auto grow to ON for **tempdb**.

Set the size of **tempdb** to an appropriate value in the production system:

* Set auto grow to ON for unplanned exceptions.
* Use instant file initialization to improve the performance of auto grow operations.

Following are some additional configuration and performance recommendations for **tempdb**:

* Create the **tempdb** database on a fast I/O subsystem. Use disk striping to numerous directly attached disks.
* Separate the disk space dedicated to **tempdb** from user databases.
* Create many files to maximize disk bandwidth and to reduce contention in allocation structures. As a general guideline, create one data file per CPU. Each file should be set to the same size. This enables the proportional fill algorithm to distribute the allocation load uniformly with minimal contention.

Use the following general guidelines when setting the FILEGROWTH increment for **tempdb**:

|  |  |
| --- | --- |
| **tempdb file size** | **FILEGROWTH increment for tempdb** |
| 0 to 100 MB | 10 MB |
| 100 to 200 MB | 20 MB |
| 500 or more | 10% |

#### Appendix: A Case Study

This case study illustrates how you can use the DMVs in SQL Server 2005 to monitor **tempdb** usage over time. An administrator can use the information gathered from monitoring to identify which version stores and/or queries are taking up the most resources in **tempdb** over time. It is then possible to take corrective action.

The case study is an example of monitoring **tempdb** space usage in a multi-user workload. This example also demonstrates the effects of versioning on **tempdb** space usage. The same test case is run under read committed isolation (the default), under read committed snapshot isolation, and under snapshot isolation.

Each action taken within the workload is encapsulated in its own stored procedure. These stored procedures are executed randomly, therefore they might not execute in the same way or in the same order in all three scenarios. While these are random, the relative weight of these stored procedures is the same across all three scenarios.

##### Machine configuration

Single Processor: 3.0 GHz

RAM: 2 GB

Drive: 100-MB SCSI

Database: 10-MB TPC-H. 1 root file, 1 database file, and 1 log file

Data Space: 29 GB

Log Space: 1 GB

OS: W2K3 SP1

Number of clients: 50

Runtime: 180 minutes prior to allowing existing connections to drain.

SQL Server used: SQL Server 2005

##### Workload

The following workloads used in the case study are similar to TPC-H workloads. The weighting that is provided is the frequency (stated as a fraction) that the driver procedure will pick a given workload to run. The times and result set sizes for the queries were measured for the initial database under single-user conditions.

sp\_takeOrder

Insertion workload. Adds an order and adds one line item.

Weighting 25/48.

sp\_deleteOrder

Deletion workload. Deletes an order and deletes any line items associated with

that order.

Weighting 10/48.

sp\_updateLineitem

Update workload. Updates a line item for a random order.

Weighting 10/48.

sp\_productTypeProfitMeasureQuery

Query workload based off of TPC-H query 9. Allocates space in **tempdb**.

Weighting 1/48.

Returns 200 rows. Takes approximately 7:00 (mm:ss) to run.

sp\_shippingModesAndOrderPriorityQuery

Query workload based off of TPC-H query 12. Allocates space in **tempdb**.

Weighting 1/48.

Returns 2 rows. Takes approximately 5:23 (mm:ss) to run.

sp\_largeVolumeCustomerQuery

Query workload based off of TPC-H query 18. Allocates space in **tempdb**.

Weighting 1/48.

Result is 14,722 rows. Takes approximately 16:53 (mm:ss) to run.

At any time during the scenario, there are 48 sessions running random workloads. Out of these 48 sessions, there are 25 active sessions executing INSERT operations. Similarly, there are 10 concurrent session doing UPDATES and DELETES. However, the queries are run in one session each. Workloads are run aggressively for an hour—when a stored procedure completes, the next is immediately initiated.

##### tempdb space usage monitoring

In this scenario, we monitor **tempdb** by using the DMVs in SQL Server 2005.

To track **tempdb** space usage over time, we periodically poll DMVs and store the output that is generated by each DMV in a database called the Perf\_warehouse. We then run various queries on the Perf\_warehouse for analysis.

Following is a sample schema that stores the output from DMVS. This is an example of tracking space in **tempdb**. You can choose a more elaborate schema that polls data from other sources such as System Monitor counters and stores the results in the performance warehouse.

###### Performance warehouse schema

CREATE database perf\_warehouse

GO

USE perf\_warehouse

GO

CREATE TABLE tempdb\_space\_usage (

  -- This represents the time when the particular row was

  -- inserted

  dt datetime DEFAULT CURRENT\_TIMESTAMP,

  -- session id of the sessions that were active at the time

  session\_id int DEFAULT null,

  -- this represents the source DMV of information. It can be

  -- track instance, session or task based allocation information.

  scope char(7),

  -- instance level unallocated extent pages in tempdb

  Instance\_unallocated\_extent\_pages bigint,

  -- tempdb pages allocated to verstion store

  version\_store\_pages bigint,

  -- tempdb pages allocated to user objects in the instance

  Instance\_userobj\_alloc\_pages bigint,

  -- tempdb pages allocated to internal objects in the instance

  Instance\_internalobj\_alloc\_pages bigint,

  -- tempdb pages allocated in mixed extents in the instance

  Instance\_mixed\_extent\_alloc\_pages bigint,

  -- tempdb pages allocated to user obejcts within this sesssion or task.

  Sess\_task\_userobj\_alloc\_pages bigint,

  -- tempdb user object pages deallocated within this sesssion

  -- or task.

  Sess\_task\_userobj\_deallocated\_pages bigint,

  -- tempdb pages allocated to internal objects within this sesssion

  -- or task

  Sess\_task\_internalobj\_alloc\_pages bigint,

  -- tempdb internal object pages deallocated within this sesssion or

  -- task

  Sess\_task\_internalobj\_deallocated\_pages bigint,

  -- query text for the active query for the task

  query\_text nvarchar(max)

)

-- Create a clustered index on time column when the data was collected

CREATE CLUSTERED INDEX cidx ON tempdb\_space\_usage (dt)

###### tempdb space usage collection

The following stored procedure polls DMVs and inserts data into the table called **tempdb\_space\_usage**. You can run this procedure as often as necessary. For our workload, we run it every minute.

CREATE PROC sp\_sampleTempDbSpaceUsage AS

  Instance level tempdb File space usage for all files within

  -- tempdb

  INSERT tempdb\_space\_usage (

    scope,

    Instance\_unallocated\_extent\_pages,

    version\_store\_pages,

    Instance\_userobj\_alloc\_pages,

    Instance\_internalobj\_alloc\_pages,

    Instance\_mixed\_extent\_alloc\_pages)

  SELECT

    'instance',

    SUM(unallocated\_extent\_page\_count),

    SUM(version\_store\_reserved\_page\_count),

    SUM(user\_object\_reserved\_page\_count),

    SUM(internal\_object\_reserved\_page\_count),

    SUM(mixed\_extent\_page\_count)

  FROM sys.dm\_db\_file\_space\_usage

    -- 2. tempdb space usage per session

    --

  INSERT tempdb\_space\_usage (

    scope,

    session\_id,

    Sess\_task\_userobj\_alloc\_pages,

    Sess\_task\_userobj\_deallocated\_pages,

    Sess\_task\_internalobj\_alloc\_pages,

    Sess\_task\_internalobj\_deallocated\_pages)

  SELECT

    'session',

    session\_id,

    user\_objects\_alloc\_page\_count,

    user\_objects\_dealloc\_page\_count,

    internal\_objects\_alloc\_page\_count,

    internal\_objects\_dealloc\_page\_count

  FROM sys.dm\_db\_session\_space\_usage

    WHERE session\_id > 50

    -- 3. tempdb space usage per active task

    --

  INSERT tempdb\_space\_usage (

    scope,

    session\_id,

    Sess\_task\_userobj\_alloc\_pages,

    Sess\_task\_userobj\_deallocated\_pages,

    Sess\_task\_internalobj\_alloc\_pages,

    Sess\_task\_internalobj\_deallocated\_pages,

    query\_text)

  SELECT

    'task',

    R1.session\_id,

    R1.user\_objects\_alloc\_page\_count,

    R1.user\_objects\_dealloc\_page\_count,

    R1.internal\_objects\_alloc\_page\_count,

    R1.internal\_objects\_dealloc\_page\_count,

    R3.text

  FROM sys.dm\_db\_task\_space\_usage AS R1

    LEFT OUTER JOIN

    sys.dm\_exec\_requests AS R2

    ON R1.session\_id = R2.session\_id

    OUTER APPLY sys.dm\_exec\_sql\_text(R2.sql\_handle) AS R3

  WHERE R1.session\_id > 50

##### tempdb space usage analysis

This is a test scenario based on a TPC-H database. Three of the TPC-H queries were specifically chosen to allocate space in **tempdb**. Other queries generate DML activity on the base tables.

None of these queries create objects in **tempdb**, so the user object level queries in the [tempdb space usage collection](#_tempdb_space_usage_collection) section in this paper are not important in this scenario. All of the sessions that connect to the test server run a procedure, exit, reconnect, and repeat. Therefore, session scope queries are not as important in this scenario. This might not be true in production systems, particularly those running ad-hoc queries.

We use the following queries to analyze **tempdb** space usage in our workload.

**Query1:** This query reports the maximum allocated space in **tempdb** over all the data points collected. You can use this to estimate the size requirements of **tempdb** for your workload. This query reports **tempdb** space usage at the instance level.

SELECT

  CONVERT (float, (MAX(version\_store\_pages +

      Instance\_userobj\_alloc\_pages +

      Instance\_internalobj\_alloc\_pages +

      Instance\_mixed\_extent\_alloc\_pages)))/ 128.0

    AS max\_tempdb\_allocation\_MB

FROM tempdb\_space\_usage

WHERE scope = ‘instance’

**Query2**: This query reports the average allocated space in **tempdb** over all the data points collected. This query reports average **tempdb** space usage at the instance level.

SELECT

  CONVERT (float, (AVG(version\_store\_pages +

      Instance\_userobj\_alloc\_pages +

      Instance\_internalobj\_alloc\_pages +

      Instance\_mixed\_extent\_alloc\_pages)))/ 128.0

    AS avg\_tempdb\_allocation\_MB

FROM tempdb\_space\_usage

WHERE scope = ‘instance’

**Query3**: This query computes the maximum allocated pages and the verison store size in megabytes over all the data points collected. If the amount of **tempdb** space that is allocated to the version store is large, it implies that long-running transactions are generating or consuming versions.

SELECT

  MAX(version\_store\_pages) AS max\_version\_store\_pages\_allocated,

  MAX(version\_store\_pages/128.0) AS max\_version\_store\_allocated\_space\_MB

FROM tempdb\_space\_usage

WHERE scope = ‘instance’

**Query4**: This query computes the average number of allocated pages and the verison store size in megabytes over all the data points collected.

SELECT

  AVG(version\_store\_pages) AS max\_version\_store\_pages\_allocated,

  AVG(version\_store\_pages)/ 128.0 AS max\_version\_store\_allocated\_space\_MB

FROM tempdb\_space\_usage

WHERE scope = ‘instance’

Results

**tempdb** space usage in the instance:

Maximum Average

Read Committed: 2232.375 MB 996.4296875 MB

RCSI: 2161.5 MB 1029.609375 MB

Snapshot: 2261.4375 MB 1031.2421875 MB

Version store space used:

Maximum Average

Read Committed: 0 MB 0 MB

RCSI: 2.312500 MB 1.546875 MB

Snapshot: 3.687500 MB 2.023437 MB

This particular workload incurs only small a space penalty for storing the row versions. Chances are that the spooling queries have a greater effect on **tempdb** space usage than does the isolation level.

Now, let us examine the amount of **tempdb** space used by tasks. Note that the **query\_text** field is truncated to better fit the page and NULL is ignored. Since there are no explicit user objects created in **tempdb**, we only monitor **tempdb** space used by internal objects.

**Query 5**: This query shows the top five queries that are allocating the most pages for internal objects.

SELECT top 5 MAX ((Sess\_task\_internalobj\_alloc\_pages) - (Sess\_task\_internalobj\_deallocated\_pages))

  AS Max\_Sess\_task\_allocated\_pages\_delta, query\_text

FROM tempdb\_space\_usage

WHERE scope = 'task' and session\_id > 50

GROUP BY query\_text

ORDER BY Max\_Sess\_task\_allocated\_pages\_delta DESC

**Query 6**: This query shows the top five queries that are allocating the most pages for internal objects on average.

SELECT top 5 AVG ((Sess\_task\_internalobj\_alloc\_pages) - (Sess\_task\_internalobj\_deallocated\_pages))

  AS Avg\_Sess\_task\_allocated\_pages\_delta, query\_text

FROM tempdb\_space\_usage

WHERE scope = 'task' and session\_id > 50

GROUP BY query\_text

ORDER BY Avg\_Sess\_task\_allocated\_pages\_delta DESC

Results

The following tables show the amount of **tempdb** space that was used by various stored procedures.

Read Committed:

**Query Maximum Average**

Create proc sp\_largeVolumeCustomerQuery ... 84952 6660

Create proc sp\_productTypeProfitMeasureQuery 32288 755

Create proc sp\_shippingModesAndOrderPriority… 672 15

CREATE PROC sp\_sampleTempDbSpaceUsage ... 24 3

Create proc sp\_deleteOrder ... 8 0

RCSI:

**Query Maximum Average**

Create proc sp\_largeVolumeCustomerQuery ... 85320 7847

Create proc sp\_productTypeProfitMeasureQuery 31608 209

Create proc sp\_shippingModesAndOrderPriority… 672 14

CREATE PROC sp\_sampleTempDbSpaceUsage ... 32 5

Create proc sp\_deleteOrder ... 8 0

Snapshot:

**Query Maximum Average**

Create proc sp\_largeVolumeCustomerQuery ... 83696 7024

Create proc sp\_productTypeProfitMeasureQuery 32288 268

Create proc sp\_shippingModesAndOrderPriority… 152 14

CREATE PROC sp\_sampleTempDbSpaceUsage ... 24 1

Create proc sp\_deleteOrder ... 8 0

This output can be used to identify which queries consume the most space in **tempdb** to store internal objects. In this particular scenario, a significant space penalty is not incurred when storing row versions. It is also possible that some of the variance is a function of the sampling interval.

##### Summary

This case study shows one method of monitoring workloads. The actual **tempdb** space usage that is due to RCSI or snapshot isolation is highly dependent on the specific server workloads. In our test workload, the main usage of **tempdb** is due to spooling.

To better evaluate throughput and to find out which isolation level works best for your workload, you can combine your **tempdb** space analysis with SQL Profiler traces.

For more information:

<http://www.microsoft.com/technet/prodtechnol/sql/default.mspx>

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), [how would you rate this paper](mailto:sqlfback@microsoft.com?subject=White%20paper%20feedback:%20Working%20with%20tempdb%20in%20SQL%20Server%202005)?