IDERA® | SQL Server Whitepaper

DEMYSTIFY TEMPDB PERFORMANCE AND MANAGEABILITY

BY ROBERT L. DAVIS

Clear guidance on best practices for managing tempdb to provide the optimum balance between performance, reliability, and manageability

Published: January 2015

Applies to:
SQL Server 2008, SQL Server 2008 R2, SQL Server 2012, SQL Server 2014, and SQL Server 2016

INTRODUCTION

There are many misconceptions and myths about tempdb and purported best practices are inconsistent at best. It's hard to know which advice to follow when one resource says to always do it one way and another tells you to always do it the opposite way. Many times, both resources are correct in certain situations or to a certain degree. Part of the problem is that rarely is there a single correct solution to any scenario in SQL Server. Unfortunately, many administrators will assume that because a practice worked for them in one situation that they should do the same practice in every situation.

The purpose of this whitepaper is to guide you on how to make the right decisions for managing *tempdb*; to help you determine the proper solution for any given scenario. In order to be able to make these decisions, there are several concepts that must be understood. We will take a look at what *tempdb* is and how it is used and some common major problems and how to prevent them. We will also provide some best practices on configuring and monitoring *tempdb*.

WHAT IS *TEMPDB*

The cornerstone of making smart decisions is being educated on the subject. Any deep discussion of tempdb should begin with an explanation of what tempdb is and how it is used. *Tempdb* is like the catchall for the database engine. One analogy that works well is a well-organized junk drawer.

The database engine uses tempdb for all of the following:

Temporary user objects

- Temp tables
- Table variables
- Temp procedures
- Global temp tables
- Cursors

Work tables / work files / intermediate results

- Sorts
    - *Index rebuilds*
    - *Some Group By, Order By, or Union operations*
- Spools
- Hash joins and hash aggregates
- Spills to tempdb
- Temporary LOB storage
- Returned tables for table valued functions
- Service broker caching
- Common Table Expressions (CTEs)

Version store for data modification transactions from

- Read committed snapshot isolation
- Snapshot isolation
- Online indexes
- After triggers
- Multiple Active Results Sets (MARS)

As you can see, *tempdb* is very much like a junk drawer. SQL has to keep track of everything in *tempdb*. It organizes the above objects into three groups: internal objects, external objects, and the version store. You can see how much total *tempdb* disk space is allocated to each category by querying the dynamic management view (DMV) *sys.dm_db_file_space_usage*.

```
SELECT FreePages = SUM(unallocated_extent_page_count),
FreeSpaceMB = SUM(unallocated_extent_page_count)/128.0,
VersionStorePages = SUM(version_store_reserved_page_count),
VersionStoreMB = SUM(version_store_reserved_page_count)/128.0,
InternalObjectPages = SUM(internal_object_reserved_page_count),
InternalObjectsMB = SUM(internal_object_reserved_page_count)/128.0,
UserObjectPages = SUM(user_object_reserved_page_count),
UserObjectsMB = SUM(user_object_reserved_page_count)/128.0
FROM sys.dm_db_file_space_usage;
```

TEMPDB CONTENTION AND CONFIGURATION

One of the most daunting performance problems with tempdb is contention on the allocation pages, often referred to as just *tempdb* contention. The default settings for tempdb are not sufficient for active systems. Even a moderately busy system can easily overcome the allocations pages for *tempdb*.

Tempdb contention is often confused with general blocking. In fact, the way it is generally detected is because it causes blocked processes. In severe cases, it can create a massive blocking chain that grows faster than processes can be cleared out. This results in client connections timing out which leads to a massive chain of blocked processes attempting to roll back.

However, this is not just simple blocking. *Tempdb* contention is latch contention on the allocation pages. There are three types of pages that experience this issue:

Page Free Space (PFS): Tracks the allocation status of each page and approximately how much free space it has. There is one PFS page for every 1/2 GB of data file. The first PFS page is page number 1 of the data file. The second is page 8088 and then it repeats every 8088 pages thereafter.

Global Allocation Map (GAM): Tracks which extents have been allocated. There is one GAM page for every 4 GB of data file. The first GAM page is page number 2 in the data file, the second is page number 511232, and then it repeats every 511,232 pages.

Shared Global Allocation Map (SGAM): Tracks which extents are being used as mixed (shared) extents. There is one SGAM page for every 4 GB of data file. The first SGAM page is page number 3 in the data file, the second is page number 511233, and then it repeats every 511,232 pages.

You can use the dynamic management view (DMV) *sys.dm_os_waiting_tasks* to find tasks that are waiting on a resource. Tasks waiting on PageIOLatch or PageLatch wait types are experiencing contention. The resource description points to the page that is experiencing contention and you can easily parse the resource description to get the page number. Then it's just a math problem to determine if it is an allocation page.

The formula for determining the type of page experiencing contention is:

- PFS: Page ID = 1 or Page ID % 8088
- GAM: Page ID = 2 or Page ID % 511232
- SGAM: Page ID = 3 or (Page ID – 1) % 511232

The following query will parse the resource description and determine if the contention being experienced is on allocation pages or not:

```
WITH Tasks
AS (SELECT session_id,
           wait_type,
           wait_duration_ms,
           blocking_session_id,
           resource_description,
           PageID = CAST(Right(resource_description,
       LEN(resource_description) -
           CHARINDEX(':', resource_description, 3))
              AS INT)
```

```
    FROM sys.dm_os_waiting_tasks
    WHERE wait_type LIKE 'PAGE%LATCH_%'
    AND resource_description LIKE '2:%')
SELECT session_id,
    wait_type,
    wait_duration_ms,
    blocking_session_id,
    resource_description,
    ResourceType =
        CASE
        WHEN PageID = 1 Or PageID % 8088 = 0
        THEN 'Is PFS Page'
        WHEN PageID = 2 Or PageID % 511232 = 0
        THEN 'Is GAM Page'
        WHEN PageID = 3 Or (PageID - 1) % 511232 = 0
        THEN 'Is SGAM Page'
        ELSE 'Is Not PFS, GAM, or SGAM page'
    END
FROM Tasks;
```

DATA FILE PER CPU CORE RATIO

The key to avoiding the problem of *tempdb* contention is proper configuration of *tempdb*. The correct number of data files for *tempdb* is a highly debated topic. For many years, Microsoft officially recommended one data file per logical CPU core (1:1 ratio), and many DBAs are familiar with this advice. Even with the most modern servers that have very high numbers of cores, this recommendation from Microsoft persisted. Many of the biggest names in SQL Server disagreed with this recommendation and advocated for a more more gradual approach to configuring the number of data files. Microsoft has recently backed off of their original recommendation and documented this different approach under KB article 2154845 (See it here).

The new recommendation is that you limit the starting point to a maximum of 8 data files. Then, if *tempdb* contention occurs, add more data files, 4 at a time, until the contention stops up to a maximum of 1 data file per logical core.

For example, if I had a system with 24 logical cores, I would start by creating a total of 8 *tempdb* data files. If *tempdb* contention occurs, I would add 4 data files bringing the total to 12. If the contention continues to occur, I would add another 4 data files and repeat as necessary until the contention was resolved or until I reached a maximum of 24 data files (1 per logical core).

Bearing in mind the change in recommendation from Microsoft, you may still find yourself asking, “Do I need one data file per logical CPU core or not?”As with most things in SQL Server, the answer is, “it depends.” Some extremely busy servers with heavy *tempdb* usage may need one data file per logical CPU core. This would qualify as one of those edge cases that somehow became the rule. Yes, it is possible that you may need one file per core, but not very likely.

It is logical to start with a ratio of one data file for every two or four cores and adjust as needed. The “as needed” part is what can kill performance of your server. As needed means that tempdb contention is occurring. This can be a minor issue if you are monitoring for *tempdb* contention and you know what action to take to alleviate it and you are comfortable with experiencing a major performance impact when it occurs until you can take steps to fix it. If you cannot answer yes to all of those stipulations, you should consider following the tried-and-true practice of one data file per core. The best practice for ratio of data files per logical CPU core can be summarized as below.

1. Create 1 *tempdb* data file per logical CPU core

OR...

1. Start With 1 data file for every 2 or 4 logical CPU cores, up to a maximum of 8
2. Monitor for *tempdb* contention
3. If *tempdb* contention is detected, increase the number of data files by 4
4. Repeat this process until contention is resolved, up to a maximum of 1 data file per logical CPU core

There will never be a need to have more than one data file per CPU core. The actual number of data files that SQL Server can use is the number of concurrent processes using *tempdb* at the same time. All processes ultimately run one task at a time on each CPU thread and there is only one CPU thread per logical CPU core. If every core is running a task and each of those tasks is using *tempdb*, the number of threads hitting *tempdb* will be the same as the number of logical cores. Your server will never have more processes than number of CPUs hitting *tempdb*.

PRE-SIZE FILES TO AVOID AUTO-GROWTH

The number of data files is only one best practice for *tempdb* that affects *tempdb* contention. The sizes of those data files are also critically important. You will often hear that *tempdb* writes to the data files in a round-robin method. In an ideal scenario, this description is sufficient. The algorithm that SQL uses to determine which file to use is actually more complex than that.

The process takes into consideration several factors. One of the main items to examine is the amount of free space on the data file. SQL also implements a clock-hand type of counter that tracks when it skips over each of the data files and when it uses them. It increments or decrements a counter based on when it uses a particular data file.

Having equal amounts of free space is critical to maintaining the round-robin data flow. It is important that the data files are all the same size and maintain the same size. The data files should be pre-sized to avoid any auto-growth of the data files. If one data file is expanded, the newly grown file will have more free space than the other data files and SQL will start writing all data to this single file until it catches up with the other files in free space. At this point, SQL may need to grow the file again. If the files are not the same size, SQL will auto-grow the largest file. You will end up with one file continually growing while the others stay small. This can cause unexpected breakouts of *tempdb* contention even though you have multiple files.

You want to avoid growing the files if at all possible. You should pre-size the data files so that they are all the same size and large enough that there won't be any need to grow the data files. My recommendation would be to pre-size all of the files, data and log, to use approximately 90% of the available drive space and disable auto-growth on the data files. I recommend allowing the log file to auto-grow by a set size, such as 512 MB, just in case. Since there should only ever be a single log file, there is very little harm if it auto-grows.

I recommend setting the log file to be approximately double the size of a single data file. The formula I recommend for calculating the size of the tempdb files given a drive with a specific size is:

```
Data file = (Size of drive * 90%) / (Number of data files + 2
    [log file will be double the data file])
Log file = Data file * 2
```

For example, if I have a 100 GB drive to use for *tempdb*, and I am creating 1 data file per CPU for an 8 CPU SQL Server, I will calculate the file sizes to be:

```
Data file = (100 GB * 90%) / (8 + 2) = 90 GB / 10 = 9 GB
Log file = 9 GB * 2 = 18 GB
```

The final calculation is 8 data files of 8 GB each and 1 log file of 18 GB leaving 10 GB free for possible log file growth.

There is another popular recommendation of implementing trace flag 1117 to force SQL Server to auto-grow all data files equally when expansion is needed to work-around this issue. The trace flag will work for *tempdb* in this situation but there is a major downside to using it. The trace flag affects every database on the server. Unlike *tempdb*, the files in a user database may not be used in the same round-robin fashion and this behavior may result in some files growing very large even though they have very little data in them.

DISK PERFORMANCE AND RAM

A lot of emphasis is placed on putting *tempdb* on the fastest disks possible. In my opinion, having plenty of RAM is even more important than disk performance.

Tempdb is being continually optimized in the engine to improve performance. One of the ways that *tempdb* performance has improved and continues to improve is its use of RAM. *Tempdb* will try to maintain objects in RAM as much as it can. There is a lot you can do to help *tempdb* stay in RAM.

First of all, you can help *tempdb* maintain objects in RAM by giving it plenty of RAM to use. If SQL is experiencing buffer pool memory pressure, SQL may trim some areas of the buffer pool to make more available to other areas such as the data cache or procedure cache.

The second thing you can do is to maintain a highly optimized system with current and meaningful statistics to avoid a phenomenon called query spills.When compiling a query, the query optimizer will calculate the amount of memory it thinks the query will need for sort, spool, or hash operations. Queries will only use a contiguous block of memory for these operations. If the amount of memory granted to the query is insufficient, the process will spill to *tempdb*. It will never request additional memory as there is no way for it to ensure that the memory will be contiguous.

When a spill occurs, everything that has been written to memory will be flushed to disk. This operation is very disk intensive on *tempdb* and can cause major blocking or delays in processing of the query.

Spills to *tempdb* may be caused by statistics not being maintained, by parameter sniffing (which results in discrepancies in the number of estimated rows versus the actual number of rows), and by widely skewed data (making it difficult for the optimizer to estimate the number of rows). In my experience, index and column statistics are the leading contributors to this problem. You can identify a currently executing spill to *tempdb* using the DMV *sys.dm_os_waiting_tasks*. The wait types most associated with a spill to *tempdb* are *IO_COMPLETION* and *SLEEP_TASK*. *IO_COMPLETION* literally means what the name implies; it is waiting for a disk IO task to complete. *SLEEP_TASK* is a generic wait type that can be seen in many different scenarios, but in most scenarios other than a spill to *tempdb*, the wait is held for such a short time that you will rarely catch it in progress. If you see sustained *SLEEP_TASK* waits, it is very likely a spill to *tempdb*.

Some common tactics for preventing these problems include using a higher sample rate for statistics updates and updating statistics more frequently. For the last case where data is widely skewed, the problem may persist even with the best possible statistics. Statistics are limited to 200 entries to try to describe the entire dataset in the index or columns. The greater the number of distinct values that must be represented, the harder it is to represent them accurately. Sometimes you can help the optimizer by adding statistics for a column that is not indexed.

A known instance of data skew can be worked around by applying a filter to statistics. If you created a filtered index, the optimizer may or may not be able to use the filtered index. However, it would be able to take advantage of the statistics created to support the filtered index. Statistics are much lighter weight than indexes and if you only need the statistics for the optimizer to get accurate counts, then it makes sense to only create the statistics.

If spills are currently occurring, you can use the DMV *sys.dm_exec_query_memory_grants* to see how much memory the optimizer thought the query needed. While the query is still running, the *ideal_memory_kb* column will show how much physical memory the optimizer thought the query would need.

You can see how much buffer pool space *tempdb* is using by querying the system catalogs *sys.allocation_units* and *sys.partitions*. The below query provides some raw details of the *tempdb* objects in the buffer pool.

```
USE tempdb;

SELECT ObjectName = OBJECT_NAME(object_id),
    object_id,
    index_id,
    allocation_unit_id,
    used_pages,
    AU.type_desc
FROM sys.allocation_units AS AU
INNER JOIN sys.partitions AS P
    -- Container is hobt for in row data
    -- and row overflow data
    ON AU.container_id = P.hobt_id
        -- IN_ROW_DATA and ROW_OVERFLOW_DATA
        AND AU.type In (1, 3)
UNION ALL
SELECT ObjectName = OBJECT_NAME(object_id),
    object_id,
    index_id,
    allocation_unit_id,
    used_pages,
    AU.type_desc
FROM sys.allocation_units AS AU
INNER JOIN sys.partitions AS P
    -- Container is partition for LOB data
    ON AU.container_id = P.partition_id
        -- LOB_DATA
        AND AU.type = 2
```

You can combine these results with the DMV *sys.dm_os_buffer_descriptors* to dig deeper into the *tempdb* objects in the buffer pool. The full query and description of the output can be found in the blog post IO — Where Are My *TempDB* Objects?

CONTENTION ON SYSTEM OBJECTS

There is a special type of contention in *tempdb* that cannot be alleviated by adding additional data files. Heavy *PageLatch* or *PageIOLatch* contention on system tables can be caused by heavy use of ad hoc temporary objects, views, or *SELECT INTO* statements. Because the contention is occurring on objects rather than allocation pages, the number of data files will not come into play. System objects are not going to be spread across files. They will be in the primary data file. The most common scenario is for the table *sys.sysmultiobjrefs*, but that does not mean it can be limited only to this table.

When you encounter heavy contention on a non-allocation page in the low 100's or lower, you may be experiencing this special form of contention. Typically, we see the resource reported as page 103 in file 1 (resource description 2:1:103 in the form <database ID>:<file ID>:<page ID>). You can confirm that the page is a system table by using DBCC PAGE to dump the page header info. When using DBCC PAGE, you will also want to turn on trace flag 3604 to output the trace info to the console like the following example.

```
DBCC TRACEON(3604);
DBCC PAGE (2, 1, 103, 1);
DBCC TRACEOFF(3604);
```

In the header section of the output, you will see a name/value pair with the name "Metadata: ObjectId". The value reported in this pair is the object ID of the table experiencing the contention. You can use this value along with the *OBJECTNAME()* and *OBJECTPROPERTYEX()* functions to see the table's name and whether or not it is a system table. For example, if the object ID is reported as 75 for the object to which the page belongs, I would execute the following command:

```
Use tempdb;
     Select TableName = OBJECT_SCHEMA_NAME(75) +
     '.' + OBJECT OBJECT_NAME(75),
     IsSystemTable = OBJECTPROPERTYEX(75, 'IsSystemTable');
```

TableName	IsSystemTable
sys.sysmultiobjrefs	1

To simplify this process, I wrote a query that will perform all the steps outlined above and return the final results. If you are getting a different wait resource than 2:1:103, be sure to change the wait resource near the top of the script.

```
Use tempdb;
Declare @WaitResource nvarchar(20) = N'2:1:103',
@DbID nvarchar(10),
@FileID nvarchar(10),
@PageID nvarchar(10),
@SQL nvarchar(100),
@ObjID int;
```

```
Declare @DBCCPage Table (ParentObject varchar(100),
     [Object] varchar(100),
     Field varchar(100),
     Value varchar(100));

Set @DbID = Left(@WaitResource, CharIndex(':', @WaitResource) - 1)
Set @FileID = SubString(@WaitResource, Len(@DatabaseID) + 2,
     CharIndex(':', @WaitResource, Len(@DbID) - 1) - 1)
Set @PageID = Right(@WaitResource,
     Len(@WaitResource) - CharIndex(':', Reverse(@WaitResource)))
Set @SQL = N'DBCC PAGE(' + @DbID + ', ' + @FileID + ', ' +
     @PageID + ', 1) WITH TABLERESULTS;';

Insert Into @DBCCPage
Exec sp_executesql @SQL;

Select @ObjID = Value
From @DBCCPage
Where ParentObject = 'PAGE HEADER:'
And Field = 'Metadata: ObjectId';

Select TableName = OBJECT_SCHEMA_NAME(@ObjID) +
     '.' + OBJECT_NAME(@ObjID),
     IsSystemTable = OBJECTPROPERTYEX(@ObjID, 'IsSystemTable');
```

Using the given wait resource, the output is once again for the table *sys.sysmultiobjrefs*.

TableName	IsSystemTable
sys.sysmultiobjrefs	1

If you are seeing contention on *sys.sysmultiobjrefs*, the only way to alleviate the contention is to modify the offending query to reduce the usage of temporary objects. One approach is to optimize how you use the temporary objects. Minimize the DDL operations on temp tables, for example, and include constraints and keys as part of the *CREATE* statement rather than doing them in different operations whenever possible. More importantly, try to remove the use of temp tables or table variables where it makes sense.

VERSION STORE MONITORING

The version store is a fairly new concept to *tempdb* administration. The version store keeps copies of data (old "versions" of the data) in *tempdb* for use by various SQL Server features, including:

- Read committed snapshot isolation
- Snapshot isolation
- Online indexes
- After triggers
- Multiple Active Results Sets (MARS)

SQL attempts to keep the version store as clean as possible. One of the misunderstood concepts of version store cleanup is that it does not de-allocate versions as soon as they are no longer being used. SQL will only clean up versions that are older than the oldest active transaction in the version store. A long running transaction can easily cause the version store to grow very large because unused versions must be maintained as long as an older transaction than itself exists.

The DMV *sys.dm_tran_active_snapshot_database_transactions* returns information about all active transactions that generate or may potentially access a version in the version store, not including system transactions. You can query this DMV for the maximum *elapsed_time_seconds* column to see the oldest transaction in the version store.

```
SELECT TOP (1) *
FROM sys.dm_tran_active_snapshot_database_transactions
ORDER BY elapsed_time_seconds DESC;
```

Key performance counters to watch for the version store are the Version Generation rate (KB/s), Version Cleanup rate (KB/s), and Version Store Size (KB) counters in the SQLServer:Transactions object. You can compare generation rate to cleanup rate to determine if the version store is growing faster than it is being cleaned up. These values can fluctuate considerably based on activity and length of transactions. The version store size should also be considered when evaluating the generation and cleanup rate. If the store size is not growing, then a high generation rate may not be much of a concern. Likewise, a really low cleanup rate may not be worth worrying about if the size is very low.

The recommendation is to baseline these counters and determine what is the normal functional range for your system when it is healthy. Then you can determine at what point the numbers may be indicative of a problem. The query on the next page will allow you to capture all three counters at the same time.

```
SELECT object_name AS 'Counter Object',
    [Version Generation rate (KB/s)],
    [Version Cleanup rate (KB/s)],
    [Version Store Size (KB)]
FROM (SELECT object_name,
        counter_name,
        cntr_value
    FROM sys.dm_os_performance_counters
    WHERE object_name = 'SQLServer:Transactions'
    AND counter_name IN (
        'Version Generation rate (KB/s)',
        'Version Cleanup rate (KB/s)',
```

```
        'Version Store Size (KB)')) AS P
PIVOT (MIN(cntr_value)
    FOR counter_name IN (
        [Version Generation rate (KB/s)],
        [Version Cleanup rate (KB/s)],
        [Version Store Size (KB)])) AS Pvt;
```

EAGER WRITES AND SQL 2014

Eager writes occur when a transaction is writing a large amount of data to the buffer pool. When a large amount of pages have been written, the eager writer process will proactively write the buffer pages to disk to minimize the load on the lazy writer and checkpoint processes. In normal operations this spreads the IO for large operations out over a longer period of time rather than having to do it all after the process has completed.

SQL Server 2014 introduced a new enhancement to eliminate eager writes for minimally logged transactions in *tempdb*. Minimally logged transactions in *tempdb* affected by this improvement include things like bulk import or *SELECT INTO* statements into a temporary table or table variable or index creation or rebuilds that use *SORT_IN_TEMPDB = ON*. Data in *tempdb* is considered volatile and is generally expected to be dropped by the end of the transaction. There is generally no need to write the data for these operations to disk as they do not need to be persisted.

Note: Here are more details on minimally logged operations.

The elimination of eager writes means that these large operations can now be performed solely in memory with no writes to disk. Disk IO is quite often the slowest part of a transaction that involves reading or writing to or from disk. You can see some considerable performance improvements in the time it takes to perform the exact same operations.

To demonstrate this, I decide to test a *SELECT INTO* statement into a temporary table using the AdventureWorksDW database in SQL Server 2012 and SQL Server 2014. I wanted to make sure the databases were identical, so I backed up the AdventureWorksDW2012 database on my SQL Server 2012 instance and restored it onto my SQL Server 2014 instance.

I identified the largest table in the database via *sys.partitions*, and I pump up the data volume many times over. I started off with a little more than 76,000 rows of data in the table *dbo.FactProductInventory* and executed the query below several times until I had almost 25 million rows. Due to a foreign key dependency, I also had to insert data into *dbo.DimProduct* as well.

```
Declare @MaxProductKey int;

-- Get maximum ProductKey (identity) value
Select @MaxProductKey = Max(ProductKey)
From dbo.DimProduct;

-- Enable identity insert to ensure no gaps exist
Set Identity_Insert dbo.DimProduct On;
```

```
-- Insert new products into dbo.DimProduct
Insert Into dbo.DimProduct (ProductKey,
     ProductAlternateKey,
     EnglishProductName,
     SpanishProductName,
     FrenchProductName,
     FinishedGoodsFlag,
     Color,
     StartDate)
Select ProductKey + @MaxProductKey,
     ProductAlternateKey,
     EnglishProductName,
     SpanishProductName,
     FrenchProductName,
     FinishedGoodsFlag,
     Color,
     StartDate = DateAdd(second, ProductKey + @MaxProductKey, getdate())
From dbo.DimProduct;

-- Disable identity insert
Set Identity_Insert dbo.DimProduct Off;

-- Insert new product inventory for new products added above
Insert Into dbo.FactProductInventory
Select ProductKey + @MaxProductKey,
     DateKey,
     MovementDate,
     UnitCost,
     UnitsIn,
     UnitsOut,
     UnitsBalance
From dbo.FactProductInventory;
```

Now that the table has more data (24,841,152 rows in my database), I run a *SELECT INTO* command into a temp table and observe the time difference using *STATISTICS IO*. The query I execute on both servers is below.

```
Set Statistics Time On;

Select *
Into #FactProductInventory
From dbo.FactProductInventory;

Set Statistics Time Off;

Drop table #FactProductInventory;
```

The relevant results from the SQL Server 2012 instance were 10.347 seconds total time:

```
SQL Server Execution Times:
     CPU time = 9500 ms, elapsed time = 10347 ms.

(24841152 row(s) affected)
```

The relevant results from the SQL Server 2014 instance were 3.665 seconds total time:

```
SQL Server Execution Times:
     CPU time = 22750 ms, elapsed time = 3665 ms.

(24841152 row(s) affected)
```

That's an improvement in execution time of about 300%. If you are doing bulk transactions like this in *tempdb* or doing index operations with sort in *tempdb*, you should see an automatic performance improvement by simply upgrading to SQL Server 2014. And if you are not using sort in *tempdb* for your index operations, you have even more reason to do so in SQL Server 2014.

One thing to bear in mind is that this enhancement has been described as a hidden performance improvement as it was not publicized or documented by the SQL Server product team. This is not an unusual thing. Every new version of SQL Server has optimizations in that do not get publicized. This is one reason why I advocate for being on the most recent major version of SQL Server as possible. You never know what performance improvement is out there that can make a huge difference in your workload.

LOCAL *TEMPDB* IN A FAILOVER CLUSTER INSTANCE

For many major versions, when running a Failover Cluster Instance (FCI) *tempdb* database files had to be located on cluster shared disks. Beginning with SQL Server 2012, you have the option to put the *tempdb* files on local, non-shared drives. Due to the volatility of *tempdb* data, there is no logical reason that a failover cluster node would need to have *tempdb* traffic shared. When a failover occurs and *tempdb* starts up, *tempdb* will be cleared and any previous objects or data will be lost.

By moving the *tempdb* files to a local disk, it greatly reduces the amount of traffic that must be transferred to the SAN or whatever shared storage system is being used. This reduces the amount of data being transferred and on busy systems can make a big difference in the performance of throughout to storage.

Additionally, if you are moving *tempdb* to local drives, you can take advantage of one of the solid-state-drive (SSD) offerings via PCI. This also provides ultra-high disk performance and throughput to the disk for *tempdb*. Even though we want *tempdb* to use RAM as much as possible, if you do run into an issue where *tempdb* writes to disk, such as a spill to *tempdb*, then the performance difference will be considerable.

When using local SSD drives, you do still want to use RAID for the drives to protect from single disk failures. It's key to remember that if your *tempdb* drive goes offline, the whole instance goes offline. Even if you only use a simple RAID mirror, it will greatly reduce the chances of *tempdb* going offline.

CONCLUSION

Tempdb is a very critical component of your SQL Server and can be a major performance pain point if not managed properly. Having a performant *tempdb* starts with proper configuration and includes consistent base lining and monitoring of *tempdb*.

- ***Tempdb* configuration:**
  - Multiple data files pre-sized equally to avoid auto-growth
    A. Use 1 data file per CPU if you are not comfortable with any part of the alternative. Or...
    B. Start with 1 data file per 2 or 4 CPUs, monitor for *tempdb* contention, and adjust the number of data files as needed
  - Pre-size the data and log files to use 90% of the available disk space
  - The log file should be twice the size of a single data file
  - Disable auto-growth of the data files
  - Set auto-growth of the log file to a hard value such as 512 MB
- **Provide plenty of RAM for *tempdb* to use**
- **Make sure queries and table statistics are optimized to avoid *tempdb* spills**
- **Monitor for *tempdb* contention**
- **Monitor the version store for long running transactions**
- **Baseline the version store to get a functional range for your server and monitor for changes outside of this range**
- **Put *tempdb* files on local SSD drives in a RAID configuration for a Failover Cluster Instance**
- **Consider upgrading to SQL 2014 if you do bulk transactions in *tempdb* or use *sort_in_tempdb* for index operations**

These best practices will help you prevent, detect, and mitigate common performance problems. Couple these best practices with an understanding of how *tempdb* works and the role it plays will help guide you to troubleshooting *tempdb* issues.

ABOUT THE AUTHOR:

Robert L Davis is a senior database administrator and technical lead at Microsoft. He is a speaker and a trainer as well as a writer for SQL Server Magazine, & co-authored “Pro SQL Server 2008 Mirroring” (Apress).

Blog: www.sqlsoldier.com **Twitter: @sqlsoldier**

IDERA

IDERA understands that IT doesn’t run on the network – it runs on the data and databases that power your business. That’s why we design our products with the database as the nucleus of your IT universe.

Our database lifecycle management solutions allow database and IT professionals to design, monitor and manage data systems with complete confidence, whether in the cloud or on-premises.

We offer a diverse portfolio of free tools and educational resources to help you do more with less while giving you the knowledge to deliver even more than you did yesterday.

Whatever your need, IDERA has a solution.

SQL Diagnostic Manager

24x7 SQL Performance Monitoring, Alerting and Diagnostics

Easily monitor and view the performance of both physical server and VMware or Hyper-V virtual server environments to get a complete view of SQL Server databases. View performance metrics and alerts for virtual machines and their related hosts including CPU, memory, disk usage, network etc. to get a complete performance picture of the SQL server environment.

- Performance monitoring for physical and virtual SQL Servers
- Deep query analysis to identify excessive waits and resource consumption
- History browsing to find and troubleshoot past issues
- Adaptive & automated alerting with 100+ pre-defined and configurable aler
- Capacity planning to see database growth trends and minimize server spraw
- SCOM management pack for integration with System Center

Start for FREE

To learn more visit idera.com today!