**Troubleshooting Performance Problems in SQL Server 2005**

Published: October 1, 2005

**Writers:** Sunil Agarwal, Boris Baryshnikov, Tom Davidson, Keith Elmore, Denzil Ribeiro, Juergen Thomas

**Applies To:** SQL Server 2005

**Summary:** It is not uncommon to experience the occasional slow down of a SQL Server database. A poorly designed database or a system that is improperly configured for the workload are but several of many possible causes of this type of performance problem. Administrators need to proactively prevent or minimize problems and, when they occur, diagnose the cause and take corrective actions to fix the problem. This paper provides step-by-step guidelines for diagnosing and troubleshooting common performance problems by using publicly available tools such as SQL Server Profiler, System Monitor, and the new Dynamic Management Views in SQL Server 2005.

**On This Page**

[Introduction](http://technet.microsoft.com/en-us/library/cc966540.aspx#ELAA)   
[Goals](http://technet.microsoft.com/en-us/library/cc966540.aspx#EKAA)   
[Methodology](http://technet.microsoft.com/en-us/library/cc966540.aspx#EJAA)   
[Resource Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#EIAA)   
[CPU Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#EHAA)   
[Memory Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#EGAA)   
[I/O Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#EFAA)   
[Tempdb](http://technet.microsoft.com/en-us/library/cc966540.aspx#EEAA)   
[Slow-Running Queries](http://technet.microsoft.com/en-us/library/cc966540.aspx#EDAA)   
[Conclusion](http://technet.microsoft.com/en-us/library/cc966540.aspx#ECAA)   
[Appendix A: DBCC MEMORYSTATUS Description](http://technet.microsoft.com/en-us/library/cc966540.aspx#EBAA)   
[Appendix B: Blocking Scripts](http://technet.microsoft.com/en-us/library/cc966540.aspx#EAAA)

**Introduction**

Many customers can experience an occasional slow down of their SQL Server database. The reasons can range from a poorly designed database to a system that is improperly configured for the workload. As an administrator, you want to proactively prevent or minimize problems and, when they occur, diagnose the cause and, when possible, take corrective actions to fix the problem. This white paper limits its scope to the problems commonly seen by Customer Support Services (CSS or PSS) at Microsoft® Corporation since an exhaustive analysis of all possible problems is not feasible. We provide step-by-step guidelines for diagnosing and troubleshooting common performance problems by using publicly available tools such as SQL Server Profiler, System Monitor (Perfmon), and the new Dynamic Management Views in Microsoft SQL Server™ 2005.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**Goals**

The primary goal of this paper is to provide a general methodology for diagnosing and troubleshooting SQL Server performance problems in common customer scenarios by using publicly available tools.

SQL Server 2005 has made great strides in supportability. The kernel layer (SQL-OS) has been re-architected and internal structures and statistical data are exposed as relational rowsets through dynamic management views (DMVs). SQL Server 2000 exposes some of this information though system tables such as **sysprocesses**, but sometimes you need to generate a physical dump of the SQL Server process memory to extract relevant information from internal structures. There are two main issues with this. First, customers cannot always provide the physical dump due to the size of the dump and the time it takes to create it. Second, it can take longer to diagnose the problem because the files must generally be transmitted to Microsoft Corporation for analysis.

This brings us to the secondary goal of this paper, which is to showcase DMVs. DMVs can expedite the diagnosis process by eliminating the need to generate and analyze physical dumps in most cases. This paper provides, when possible, a side-by-side comparison of troubleshooting the same problem in SQL Server 2000 and in SQL Server 2005. DMVs provide a simplified and familiar relational interface for getting critical system information. This information can be used for monitoring purposes to alert administrators to any potential problems. Or, the information can be polled and collected periodically for detailed analysis later.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**Methodology**

There can be many reasons for a slowdown in SQL Server. We use the following three key symptoms to start diagnosing problems.

* **Resource bottlenecks**: CPU, memory, and I/O bottlenecks are covered in this paper. We do not consider network issues. For each resource bottleneck, we describe how to identify the problem and then iterate through the possible causes. For example, a memory bottleneck can lead to excessive paging that ultimately impacts performance.
* **Tempdb bottlenecks**: Since there is only one **tempdb** for each SQL Server instance, this can be a performance and a disk space bottleneck. A misbehaving application can overload **tempdb** both in terms of excessive DDL/DML operations and in space. This can cause unrelated applications running on the server to slow down or fail.
* **A slow running user query**: The performance of an existing query may regress or a new query may appear to be taking longer than expected. There can be many reasons for this. For example:
  + Changes in statistical information can lead to a poor query plan for an existing query.
  + Missing indexes can force table scans and slow down the query.
  + An application can slow down due to blocking even if resource utilization is normal.

Excessive blocking, for example, can be due to poor application or schema design or choosing an improper isolation level for the transaction.

The causes of these symptoms are not necessarily independent of each other. The poor choice of a query plan can tax system resources and cause an overall slowdown of the workload. So, if a large table is missing a useful index, or the query optimizer decides not to use it, this not only causes the query to slow down but it also puts heavy pressure on the I/O subsystem to read the unnecessary data pages and on the memory (buffer pool) to store these pages in the cache. Similarly, excessive recompilation of a frequently running query can put pressure on the CPU.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**Resource Bottlenecks**

The next sections of this paper discuss the CPU, memory, and I/O subsystem resources and how these can become bottlenecks. (Network issues are outside of the scope of this paper.) For each resource bottleneck, we describe how to identify the problem and then iterate through the possible causes. For example, a memory bottleneck can lead to excessive paging, which can ultimately impact performance.

Before you can determine if you have a resource bottleneck, you need to know how resources are used under normal circumstances. You can use the methods outlined in this paper to collect baseline information about the use of the resource (when you are not having performance problems).

You might find that the problem is a resource that is running near capacity and that SQL Server cannot support the workload in its current configuration. To address this issue, you may need to add more processing power, memory, or increase the bandwidth of your I/O or network channel. But, before you take that step, it is useful to understand some common causes of resource bottlenecks. There are solutions that do not require adding additional resources as, for example, reconfiguration.

**Tools for resolving resource bottlenecks**

One or more of the following tools are used to resolve a particular resource bottleneck.

* **System Monitor (PerfMon)**: This tool is available as part of Windows. For more information, please see the System Monitor documentation.
* **SQL Server Profiler**: See **SQL Server Profiler** in the **Performance Tools** group in the **SQL Server 2005** program group.
* **DBCC commands**: See SQL Server Books Online and Appendix A for details.
* **DMVs**: See SQL Server Books Online for details.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**CPU Bottlenecks**

A CPU bottleneck that happens suddenly and unexpectedly, without additional load on the server, is commonly caused by a nonoptimal query plan, a poor configuration, or design factors, and not insufficient hardware resources. Before rushing out to buy faster and/or more processors, you should first identify the largest consumers of CPU bandwidth and see if they can be tuned.

System Monitor is generally the best means to determine if the server is CPU bound. You should look to see if the **Processor:% Processor Time** counter is high; values in excess of 80% processor time per CPU are generally deemed to be a bottleneck. You can also monitor the SQL Server schedulers using the sys.dm\_os\_schedulers view to see if the number of runnable tasks is typically nonzero. A nonzero value indicates that tasks have to wait for their time slice to run; high values for this counter are a symptom of a CPU bottleneck. You can use the following query to list all the schedulers and look at the number of runnable tasks.

select

    scheduler\_id,

    current\_tasks\_count,

    runnable\_tasks\_count

from

    sys.dm\_os\_schedulers

where

    scheduler\_id < 255

The following query gives you a high-level view of which currently cached batches or procedures are using the most CPU. The query aggregates the CPU consumed by all statements with the same plan\_\_handle (meaning that they are part of the same batch or procedure). If a given plan\_handle has more than one statement, you may have to drill in further to find the specific query that is the largest contributor to the overall CPU usage.

select top 50

    sum(qs.total\_worker\_time) as total\_cpu\_time,

    sum(qs.execution\_count) as total\_execution\_count,

    count(\*) as  number\_of\_statements,

    qs.plan\_handle

from

    sys.dm\_exec\_query\_stats qs

group by qs.plan\_handle

order by sum(qs.total\_worker\_time) desc

The remainder of this section discusses some common CPU-intensive operations that can occur with SQL Server, as well as efficient methods to detect and resolve these problems.

**Excessive compilation and recompilation**

When a batch or remote procedure call (RPC) is submitted to SQL Server, before it begins executing the server checks for the validity and correctness of the query plan. If one of these checks fails, the batch may have to be compiled again to produce a different query plan. Such compilations are known as *recompilations*. These recompilations are generally necessary to ensure correctness and are often performed when the server determines that there could be a more optimal query plan due to changes in underlying data. Compilations by nature are CPU intensive and hence excessive recompilations could result in a CPU-bound performance problem on the system.

In SQL Server 2000, when SQL Server recompiles a stored procedure, the entire stored procedure is recompiled, not just the statement that triggered the recompile. SQL Server 2005 introduces statement-level recompilation of stored procedures. When SQL Server 2005 recompiles stored procedures, only the statement that caused the recompilation is compiled—not the entire procedure. This uses less CPU bandwidth and results in less contention on lock resources such as COMPILE locks. Recompilation can happen due to various reasons, such as:

* Schema changed
* Statistics changed
* Deferred compile
* SET option changed
* Temporary table changed
* Stored procedure created with the RECOMPILE query hint or which uses OPTION (RECOMPILE)

**Detection**

You can use System Monitor (PerfMon) or SQL Trace (SQL Server Profiler) to detect excessive compiles and recompiles.

**System Monitor (Perfmon)**

The **SQL Statistics** object provides counters to monitor compilation and the type of requests that are sent to an instance of SQL Server. You must monitor the number of query compilations and recompilations in conjunction with the number of batches received to find out if the compiles are contributing to high CPU use. Ideally, the ratio of **SQL Recompilations/sec** to **Batch Requests/sec** should be very low unless users are submitting ad hoc queries.

The key data counters to look are as follows.

* SQL Server: **SQL Statistics: Batch Requests/sec**
* SQL Server: **SQL Statistics: SQL Compilations/sec**
* SQL Server: **SQL Statistics: SQL Recompilations/sec**

For more information, see “SQL Statistics Object” in SQL Server Books Online.

**SQL Trace**

If the PerfMon counters indicate a high number of recompiles, the recompiles could be contributing to the high CPU consumed by SQL Server. We would then need to look at the profiler trace to find the stored procedures that were being recompiled. The SQL Server Profiler trace gives us that information along with the reason for the recompilation. You can use the following events to get this information.

**SP:Recompile / SQL:StmtRecompile**. The **SP:Recompile** and the **SQL:StmtRecompile** event classes indicate which stored procedures and statements have been recompiled. When you compile a stored procedure, one event is generated for the stored procedure and one for each statement that is compiled. However, when a stored procedure recompiles, only the statement that caused the recompilation is recompiled (not the entire stored procedure as in SQL Server 2000). Some of the more important data columns for the **SP:Recompile** event class are listed below. The **EventSubClass** data column in particular is important for determining the reason for the recompile. **SP:Recompile** is triggered once for the procedure or trigger that is recompiled and is not fired for an ad hoc batch that could likely be recompiled. In SQL Server 2005, it is more useful to monitor **SQL:StmtRecompiles** as this event class is fired when any type of batch, ad hoc, stored procedure, or trigger is recompiled.

The key data columns we look at in these events are as follows.

* EventClass
* EventSubClass
* ObjectID (represents stored procedure that contains this statement)
* SPID
* StartTime
* SqlHandle
* TextData

For more information, see “SQL:StmtRecompile Event Class” in SQL Server Books Online.

If you have a trace file saved, you can use the following query to see all the recompile events that were captured in the trace.

select

    spid,

    StartTime,

    Textdata,

    EventSubclass,

    ObjectID,

    DatabaseID,

    SQLHandle

from

    fn\_trace\_gettable ( 'e:\recompiletrace.trc' , 1)

where

    EventClass in(37,75,166)

EventClass  37 = Sp:Recompile, 75 = CursorRecompile, 166=SQL:StmtRecompile

You could further group the results from this query by the SqlHandle and ObjectID columns, or by various other columns, in order to see if most of the recompiles are attributed by one stored procedure or are due to some other reason (such as a SET option that has changed).

**Showplan XML For Query Compile**. The **Showplan XML For Query Compile** event class occurs when Microsoft SQL Server compiles or recompiles a SQL statement. This event has information about the statement that is being compiled or recompiled. This information includes the query plan and the object ID of the procedure in question. Capturing this event has significant performance overhead, as it is captured for each compilation or recompilation. If you see a high value for the **SQL Compilations/sec** counter in System Monitor, you should monitor this event. With this information, you can see which statements are frequently recompiled. You can use this information to change the parameters of those statements. This should reduce the number of recompiles.

**DMVs**. When you use the **sys.dm\_exec\_query\_optimizer\_info** DMV, you can get a good idea of the time SQL Server spends optimizing. If you take two snapshots of this DMV, you can get a good feel for the time that is spent optimizing in the given time period.

select \*

from sys.dm\_exec\_query\_optimizer\_info

counter          occurrence           value

---------------- -------------------- ---------------------

optimizations    81                   1.0

elapsed time     81                   6.4547820702944486E-2

In particular, look at the elapsed time, which is the time elapsed due to optimizations. Since the elapsed time during optimization is generally close to the CPU time that is used for the optimization (since the optimization process is very CPU bound), you can get a good measure of the extent to which the compile time is contributing to the high CPU use.

Another DMV that is useful for capturing this information is **sys.dm\_exec\_query\_stats**.

The data columns that you want to look at are as follows. :

* Sql\_handle
* Total worker time
* Plan generation number
* Statement Start Offset

For more information, see the SQL Server Books Online topic on **sys.dm\_exec\_query\_stats**.

In particular, plan\_generation\_num indicates the number of times the query has recompiled. The following sample query gives you the top 25 stored procedures that have been recompiled.

select \*

from sys.dm\_exec\_query\_optimizer\_info

select top 25

    sql\_text.text,

    sql\_handle,

    plan\_generation\_num,

    execution\_count,

    dbid,

    objectid

from

    sys.dm\_exec\_query\_stats a

    cross apply sys.dm\_exec\_sql\_text(sql\_handle) as sql\_text

where

    plan\_generation\_num >1

order by plan\_generation\_num desc

For additional information, see [Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005](http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx) (http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx) on Microsoft TechNet.

**Resolution**

If you have detected excessive compilation/recompilation, consider the following options.

* If the recompile occurred because a SET option changed, use SQL Server Profiler to determine which SET option changed. Avoid changing SET options within stored procedures. It is better to set them at the connection level. Ensure that SET options are not changed during the lifetime of the connection.
* Recompilation thresholds for temporary tables are lower than for normal tables. If the recompiles on a temporary table are due to statistics changes, you can change the temporary tables to table variables. A change in the cardinality of a table variable does not cause a recompilation. The drawback of this approach is that the query optimizer does not keep track of a table variable’s cardinality because statistics are not created or maintained on table variables. This can result in nonoptimal query plans. You can test the different options and choose the best one.

Another option is to use the KEEP PLAN query hint. This sets the threshold of temporary tables to be the same as that of permanent tables. The **EventSubclass** column indicates that “Statistics Changed” for an operation on a temporary table.

* To avoid recompilations that are due to changes in statistics (for example, when the plan becomes suboptimal due to change in the data statistics), specify the KEEPFIXED PLAN query hint. With this option in effect, recompilations can only happen because of correctness-related reasons (for example, when the underlying table structure has changed and the plan no longer applies) and not due to statistics. An example might be when a recompilation occurs if the schema of a table that is referenced by a statement changes, or if a table is marked with the **sp\_recompile** stored procedure.
* Turning off the automatic updates of statistics for indexes and statistics that are defined on a table or indexed view prevents recompiles that are due to statistics changes on that object. Note, however, that turning off the “auto-stats” feature by using this method is usually not a good idea. This is because the query optimizer is no longer sensitive to data changes in those objects and suboptimal query plans might result. Use this method only as a last resort after exhausting all other alternatives.
* Batches should have qualified object names (for example, dbo.Table1) to avoid recompilation and to avoid ambiguity between objects.
* To avoid recompiles that are due to deferred compiles, do not interleave DML and DDL or create the DDL from conditional constructs such as IF statements.
* Run Database Engine Tuning Advisor (DTA) to see if any indexing changes improve the compile time and the execution time of the query.
* Check to see if the stored procedure was created with the WITH RECOMPILE option or if the RECOMPILE query hint was used. If a procedure was created with the WITH RECOMPILE option, in SQL Server 2005, we may be able to take advantage of the statement level RECOMPILE hint if a particular statement within that procedure needs to be recompiled. This would avoid the necessity of recompiling the whole procedure each time it executes, while at the same time allowing the individual statement to be compiled. For more information on the RECOMPILE hint, see SQL Server Books Online.

**Inefficient query plan**

When generating an execution plan for a query, the SQL Server optimizer attempts to choose a plan that provides the fastest response time for that query. Note that the fastest response time doesn’t necessarily mean minimizing the amount of I/O that is used, nor does it necessarily mean using the least amount of CPU—it is a balance of the various resources.

Certain types of operators are more CPU intensive than others. By their nature, the **Hash** operator and **Sort** operator scan through their respective input data. With read ahead (prefetch) being used during such a scan, the pages are almost always available in the buffer cache before the page is needed by the operator. Thus, waits for physical I/O are minimized or eliminated. When these types of operations are no longer constrained by physical I/O, they tend to manifest themselves by high CPU consumption. By contrast, nested loop joins have many index lookups and can quickly become I/O bound if the index lookups are traversing to many different parts of the table so that the pages can’t fit into the buffer cache.

The most significant input the optimizer uses in evaluating the cost of various alternative query plans is the cardinality estimates for each operator, which you can see in the Showplan (**EstimateRows** and **EstimateExecutions** attributes). Without accurate cardinality estimates, the primary input used in optimization is flawed, and many times so is the final plan.

For an excellent white paper that describes in detail how the SQL Server optimizer uses statistics, see [*S*tatistics Used by the Query Optimizer in Microsoft SQL Server 200*5*](http://www.microsoft.com/technet/prodtechnol/sql/2005/qrystats.mspx) (http://www.microsoft.com/technet/prodtechnol/sql/2005/qrystats.mspx). The white paper discusses how the optimizer uses statistics, best practices for maintaining up-to-date statistics, and some common query design issues that can prevent accurate estimate cardinality and thus cause inefficient query plans.

**Detection**

Inefficient query plans are usually detected comparatively. An inefficient query plan may cause increased CPU consumption.

The query against **sys.dm\_exec\_query\_stats** is an efficient way to determine which query is using the most cumulative CPU.

select

    highest\_cpu\_queries.plan\_handle,

    highest\_cpu\_queries.total\_worker\_time,

    q.dbid,

    q.objectid,

    q.number,

    q.encrypted,

    q.[text]

from

    (select top 50

        qs.plan\_handle,

        qs.total\_worker\_time

    from

        sys.dm\_exec\_query\_stats qs

    order by qs.total\_worker\_time desc) as highest\_cpu\_queries

    cross apply sys.dm\_exec\_sql\_text(plan\_handle) as q

order by highest\_cpu\_queries.total\_worker\_time desc

Alternatively, query against **sys.dm\_exec\_cached\_plans** by using filters for various operators that may be CPU intensive, such as ‘%Hash Match%’, ‘%Sort%’ to look for suspects.

**Resolution**

Consider the following options if you have detected inefficient query plans.

* Tune the query with the Database Engine Tuning Advisor to see if it produces any index recommendations.
* Check for issues with bad cardinality estimates.

Are the queries written so that they use the most restrictive WHERE clause that is applicable? Unrestricted queries are resource intensive by their very nature.

Run UPDATE STATISTICS on the tables involved in the query and check to see if the problem persists.

Does the query use constructs for which the optimizer is unable to accurately estimate cardinality? Consider whether the query can be modified in a way so that the issue can be avoided.

* If it is not possible to modify the schema or the query, SQL Server 2005 has a new plan guide feature that allows you to specify query hints to add to queries that match certain text. This can be done for ad hoc queries as well as inside a stored procedure. Hints such as OPTION (OPTIMIZE FOR) allow you to impact the cardinality estimates while leaving the optimizer its full array of potential plans. Other hints such as OPTION (FORCE ORDER) or OPTION (USE PLAN) allow you varying degrees of control over the query plan.

**Intra-query parallelism**

When generating an execution plan for a query, the SQL Server optimizer attempts to choose the plan that provides the fastest response time for that query. If the query’s cost exceeds the value specified in the **cost threshold for parallelism** option and parallelism has not been disabled, then the optimizer attempts to generate a plan that can be run in parallel. A parallel query plan uses multiple threads to process the query, with each thread distributed across the available CPUs and concurrently utilizing CPU time from each processor. The maximum degree of parallelism can be limited server wide using the **max degree of parallelism** option or on a per-query level using the OPTION (MAXDOP) hint.

The decision on the actual degree of parallelism (DOP) used for execution—a measure of how many threads will do a given operation in parallel—is deferred until execution time. Before executing the query, SQL Server  2005 determines how many schedulers are under-utilized and chooses a DOP for the query that fully utilizes the remaining schedulers. Once a DOP is chosen, the query runs with the chosen degree of parallelism until completion. A parallel query typically uses a similar but slightly higher amount of CPU time as compared to the corresponding serial execution plan, but it does so in a shorter duration of elapsed time. As long as there are no other bottlenecks, such as waits for physical I/O, parallel plans generally should use 100% of the CPU across all of the processors.

One key factor (how idle the system is) that led to running a parallel plan can change after the query starts executing. This can change, however, after the query starts executing. For example, if a query comes in during an idle time, the server may choose to run with a parallel plan and use a DOP of four and spawn up threads on four different processors. Once those threads start executing, existing connections may submit other queries that also require a lot of CPU. At that point, all the different threads will share short time slices of the available CPU, resulting in higher query duration.

Running with a parallel plan is not inherently bad and should provide the fastest response time for that query. However, the response time for a given query must be weighed against the overall throughput and responsiveness of the rest of the queries on the system. Parallel queries are generally best suited to batch processing and decision support workloads and might not be desirable in a transaction processing environment.

**Detection**

Intra-query parallelism problems can be detected using the following methods.

**System Monitor (Perfmon)**

Look at the **SQL Server:SQL Statistics – Batch Requests/sec** counter and see “SQL Statistics Object” in SQL Server Books Online for more information.

Because a query must have an estimated cost that exceeds the cost threshold for the parallelism configuration setting (which defaults to 5) before it is considered for a parallel plan, the more batches a server is processing per second the less likely it is that the batches are running with parallel plans. Servers that are running many parallel queries normally have small batch requests per second (for example, values less than 100).

**DMVs**

From a running server, you can determine whether any active requests are running in parallel for a given session by using the following query.

select

    r.session\_id,

    r.request\_id,

    max(isnull(exec\_context\_id, 0)) as number\_of\_workers,

    r.sql\_handle,

    r.statement\_start\_offset,

    r.statement\_end\_offset,

    r.plan\_handle

from

    sys.dm\_exec\_requests r

    join sys.dm\_os\_tasks t on r.session\_id = t.session\_id

    join sys.dm\_exec\_sessions s on r.session\_id = s.session\_id

where

    s.is\_user\_process = 0x1

group by

    r.session\_id, r.request\_id,

    r.sql\_handle, r.plan\_handle,

    r.statement\_start\_offset, r.statement\_end\_offset

having max(isnull(exec\_context\_id, 0)) > 0

With this information, the text of the query can easily be retrieved by using **sys.dm\_exec\_sql\_text**, while the plan can be retrieved using **sys.dm\_exec\_cached\_plan**.

You may also search for plans that are eligible to run in parallel. This can be done by searching the cached plans to see if a relational operator has its **Parallel** attribute as a nonzero value. These plans may not run in parallel, but they are eligible to do so if the system is not too busy.

--

-- Find query plans that may run in parallel

--

select

    p.\*,

    q.\*,

    cp.plan\_handle

from

    sys.dm\_exec\_cached\_plans cp

    cross apply sys.dm\_exec\_query\_plan(cp.plan\_handle) p

    cross apply sys.dm\_exec\_sql\_text(cp.plan\_handle) as q

where

    cp.cacheobjtype = 'Compiled Plan' and

    p.query\_plan.value('declare namespace

p="http://schemas.microsoft.com/sqlserver/2004/07/showplan";

        max(//p:RelOp/@Parallel)', 'float') > 0

In general, the duration of a query is longer than the amount of CPU time, because some of the time was spent waiting on resources such as a lock or physical I/O. The only scenario where a query can use more CPU time than the elapsed duration is when the query runs with a parallel plan such that multiple threads are concurrently using CPU. Note that not all parallel queries will demonstrate this behavior (CPU time greater than the duration).

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

select

    qs.sql\_handle,

    qs.statement\_start\_offset,

    qs.statement\_end\_offset,

    q.dbid,

    q.objectid,

    q.number,

    q.encrypted,

    q.text

from

    sys.dm\_exec\_query\_stats qs

    cross apply sys.dm\_exec\_sql\_text(qs.plan\_handle) as q

where

    qs.total\_worker\_time > qs.total\_elapsed\_time

SQL Trace

Look for the following signs of parallel queries,

which could be either statements or batches that have

CPU time greater than the duration.

select

    EventClass,

    TextData

from

    ::fn\_trace\_gettable('c:\temp\high\_cpu\_trace.trc',

default)

where

    EventClass in (10, 12)    -- RPC:Completed,

SQL:BatchCompleted

    and CPU > Duration/1000    -- CPU is in

milliseconds, Duration in microseconds Or can be

Showplans (un-encoded) that have Parallelism operators]

in them

select

    EventClass,

    TextData

from

    ::fn\_trace\_gettable('c:\temp\high\_cpu\_trace.trc',

default)

where

    TextData LIKE '%Parallelism%'

**Resolution**

Any query that runs with a parallel plan is one that the optimizer believes is expensive enough that it would exceed the cost threshold of parallelism, which defaults to five (roughly 5-second execution time on a reference machine). Any queries identified through the methods above are candidates for further tuning.

* Use the Database Engine Tuning Advisor to see if any indexing changes, changes to indexed views, or partitioning changes could reduce the cost of the query.
* Check for significant differences in the actual versus the estimated cardinality since the cardinality estimates are the primary factor in estimating the cost of the query. If any significant differences are found:

If the **auto create statistics** database option is disabled, make sure that there are no MISSING STATS entries in the **Warnings** column of the Showplan output.

Try running UPDATE STATISTICS on the tables where the cardinality estimates are off.

Verify that the query doesn’t use a query construct that the optimizer can’t accurately estimate, such as multi-statement table-valued functions or CLR functions, table variables, or comparisons with a Transact-SQL variable (comparisons with a parameter are OK).

* Evaluate whether the query could be written in a more efficient fashion using different Transact-SQL statements or expressions.

**Poor cursor usage**

Versions of SQL Server prior to SQL Server 2005 only supported a single active common per connection. A query that was executing or had results pending to send to the client was considered active. In some situations, the client application might need to read through the results and submit other queries to SQL Server based on the row just read from the result set. This could not be done with a default result set, since it could have other pending results. A common solution was to change the connection properties to use a server-side cursor.

When using a server-side cursor, the database client software (the OLE DB provider or ODBC driver) transparently encapsulates client requests inside of special extended stored procedures, such as **sp\_cursoropen**, **sp\_cursorfetch**, and so forth. This is referred to as an *API cursor* (as opposed to a TSQL cursor). When the user executes the query, the query text is sent to the server via **sp\_cursoropen**, requests to read from the result set would result in an **sp\_cursorfetch** instructing the server to only send back a certain number of rows. By controlling the number of rows that are fetched, it is possible for the ODBC driver or OLE DB provider to cache the row(s). This prevents a situation where the server is waiting for the client to read all the rows it has sent. Thus, the server is ready to accept a new request on that connection.

Applications that open cursors and fetch one row (or a small number of rows) at a time can easily become bottlenecked by the network latency, especially on a wide area network (WAN). On a fast network with many different user connections, the overhead required to process many cursor requests may become significant. Because of the overhead associated with repositioning the cursor to the appropriate location in the result set, per-request processing overhead, and similar processing, it is more efficient for the server to process a single request that returns 100 rows than to process 100 separate requests which return the same 100 rows but one row at a time.

**Detection**

You can use the following methods to troubleshoot poor cursor usage.

**System Monitor (Perfmon)**

By looking at the **SQL Server:Cursor Manager By Type – Cursor Requests/Sec** counter, you can get a general feel for how many cursors are being used on the system by looking at this performance counter. Systems that have high CPU utilization because of small fetch sizes typically have hundreds of cursor requests per second. There are no specific counters to tell you about the fetch buffer size.

**DMVs**

The following query can be used to determine the connections with API cursors (as opposed to TSQL cursors) that are using a fetch buffer size of one row. It is much more efficient to use a larger fetch buffer, such as 100 rows.

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

select

    cur.\*

from

    sys.dm\_exec\_connections con

    cross apply sys.dm\_exec\_cursors(con.session\_id) as cur

where

    cur.fetch\_buffer\_size = 1

    and cur.properties LIKE 'API%'    -- API

cursor (TSQL cursors always have fetch buffer of 1)

**SQL Trace**

Use a trace that includes the **RPC:Completed** event class search for **sp\_cursorfetch** statements. The value of the fourth parameter is the number of rows returned by the fetch. The maximum number of rows that are requested to be returned is specified as an input parameter in the corresponding **RPC:Starting** event class.

**Resolution**

* Determine if cursors are the most appropriate means to accomplish the processing or whether a set-based operation, which is generally more efficient, is possible.
* Consider enabling multiple active results (MARS) when connecting to SQL Server 2005.
* Consult the appropriate documentation for your specific API to determine how to specify a larger fetch buffer size for the cursor:

ODBC - **SQL\_ATTR\_ROW\_ARRAY\_SIZE**

OLE DB – **IRowset::GetNextRows or IRowsetLocate::GetRowsAt**

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**Memory Bottlenecks**

This section specifically addresses low memory conditions and ways to diagnose them as well as different memory errors, possible reasons for them, and ways to troubleshoot.

**Background**

It is quite common to refer to different memory resources by using the single generic term *memory*. As there are several types of memory resources, it is important to understand and differentiate which particular memory resource is referred to.

**Virtual address space and physical memory**

In Microsoft Windows®, each process has its own virtual address space (VAS). The set of all virtual addresses available for process use constitutes the size of the VAS. The size of the VAS depends on the architecture (32- or 64-bit) and the operating system. In the context of troubleshooting, it is important to understand that virtual address space is a consumable memory resource and an application can run out of it even on a 64-bit platform while physical memory may still be available.

For more information about virtual address space, see “Process Address Space” in SQL Server Books Online and the article called [Virtual Address Space](http://msdn2.microsoft.com/library/aa366912.aspx) (http://msdn2.microsoft.com/en-us/library/aa366912.aspx) on MSDN.

**Address Windowing Extensions (AWE) and SQL Server**

Address Windowing Extensions (AWE) is an API that allows a 32-bit application to manipulate physical memory beyond the inherent 32-bit address limit. AWE mechanism technically is not necessary on 64-bit platform. It is, however, present there. Memory pages that are allocated through the AWE mechanism are referred as *locked pages* on the 64-bit platform.

On both 32- and 64-bit platforms, memory that is allocated through the AWE mechanism cannot be paged out. This can be beneficial to the application. (This is one of the reasons for using AWE mechanism on 64-bit platform.) This also affects the amount of RAM that is available to the system and to other applications, which may have detrimental effects. For this reason, in order to use AWE, the **Lock Pages in Memory** privilege must be enabled for the account that runs SQL Server.

From a troubleshooting perspective, an important point is that the SQL Server buffer pool uses AWE mapped memory; however, only database (hashed) pages can take full advantage of memory allocated through AWE. Memory allocated through the AWE mechanism is not reported by Task Manager or in the **Process: Private Bytes** performance counter. You need to use SQL Server specific counters or Dynamic Management Views to obtain this information.

For more information about AWE mapped memory, see “Managing memory for large databases” and “Memory Architecture” in SQL Server Books Online  topics and [Large Memory Support](http://msdn2.microsoft.com/library/aa366718.aspx) (http://msdn2.microsoft.com/library/aa366718.aspx) on MSDN.

The following table summarizes the maximum memory support options for different configurations of SQL Server 2005. (Note that a particular edition of SQL Server or Windows may put more restrictive limits on the amount of supported memory.)

**Table 1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Configuration** | **VAS** | **Max physical memory** | **AWE/locked pages support** |
| Native 32-bit on 32-bit OS  with /3GB boot parameter1 | 2 GB  3 GB | 64 GB  16 GB | Yes  Yes |
| 32-bit on x64 OS (WOW) | 4 GB | 64 GB | Yes |
| 32-bit on IA64 OS (WOW) | 2 GB | 2 GB | No |
| Native 64-bit on x64 OS | 8 terabyte | 1 terabyte | Yes |
| Native 64-bit on IA64 OS | 7 terabyte | 1 terabyte | Yes |

**Memory pressures**

Memory pressure denotes a condition when limited amount of memory is available. Identifying when SQL Server runs under a memory pressure will help you troubleshoot memory-related issues. SQL Server responds differently depending on the type of memory pressure that is present. The following table summarizes the types of memory pressures, and their general underlying causes. In all cases, you are more likely to see timeout or explicit out-of-memory error messages.

**Table 2**

|  |  |  |
| --- | --- | --- |
| **Pressure** | **External** | **Internal** |
| Physical | Physical memory (RAM) running low. This causes the system to trim working sets of currently running processes, which may result in overall slowdown.  SQL Server detects this condition and, depending on the configuration, may reduce the commit target of the buffer pool and start clearing internal caches. | SQL Server detects high memory consumption internally, causing redistribution of memory between internal components.  Internal memory pressure may be a result of:   * Responding to the external memory pressure (SQL Server sets lower memory usage caps). * Changed memory settings (e.g. ‘max server memory’). * Changes in memory distribution of internal components (due to high percentage of reserved and stolen pages from the buffer pool). |
| Virtual | Running low on space in the system page file(s). This may cause the system to fail memory allocations, as it is unable to page out currently allocated memory. This condition may result in the whole system responding very slowly or even bring it to a halt. | Running low on VAS due to fragmentation (a lot of VAS is available but in small blocks) and/or consumption (direct allocations, DLLs loaded in SQL Server VAS, high number of threads).  SQL Server detects this condition and may release reserved regions of VAS, reduce buffer pool commit target, and start shrinking caches. |

Windows has a notification mechanism2 if physical memory is running high or low. SQL Server uses this mechanism in its memory management decisions.

General troubleshooting steps in each case are explained in Table 3.

**Table 3**

|  |  |  |
| --- | --- | --- |
| **Pressure** | **External** | **Internal** |
| Physical | * Find major system memory consumers. * Attempt to eliminate (if possible). * Check for adequate system RAM and consider adding more RAM (usually requires more careful investigation beyond the scope of this paper). | * Identify major memory consumers inside SQL Server. * Verify server configuration. * Further actions depend on the investigation: check for workload; possible design issues; other resource bottlenecks. |
| Virtual | * Increase swap file size. * Check for major physical memory consumers and follow steps of external physical memory pressure. | * Follow steps of internal physical memory pressure. |

**Tools**

The following tools and sources of information could be used for troubleshooting.

* Memory related DMVs
* DBCC MEMORYSTATUS command
* Performance counters: performance monitor or DMV for SQL Server specific object
* Task Manager
* Event viewer: application log, system log

**Detecting memory pressures**

Memory pressure by itself does not indicate a problem. Memory pressure is a necessary but not a sufficient condition for the server to encounter memory errors later on. Working under memory pressure could be a normal operating condition of the server. However, signs of memory pressure may indicate that the server runs close to its capacity and the potential for out-of-memory errors exists. In the case of normally operating server, this information could serve as a baseline for determining reasons for out-of-memory conditions later.

**External physical memory pressure**

Open Task Manager in Performance view and check the **Physical Memory** section, **Available** value. If the available memory amount is low, external memory pressure may be present. The exact value depends on many factors, however you can start looking into this when the value drops below 50-100 MB. External memory pressure is clearly present when this amount is less than 10 MB.

The equivalent information can also be obtained using the **Memory: Available Bytes** counter in System Monitor.

If external memory pressure exists and you are seeing memory-related errors, you will need to identify major consumers of the physical memory on the system. To do this, look at  **Process: Working Set** performance counters or the **Mem Usage** column on the **Processes** tab of Task Manager and identify the largest consumers.

The total use of physical memory on the system can be roughly accounted for by summing the following counters.

* **Process** object, **Working Set** counter for each process
* **Memory** object
  + **Cache Bytes** counter for system working set
  + **Pool Nonpaged Bytes** counter for size of unpaged pool
  + **Available Bytes** (equivalent of the **Available** value in Task Manager)

If there’s no external pressure, the **Process: Private Bytes** counter or the VM Size in Task Manager should be close to the size of the working set (**Process: Working Set** or Task Manager **Mem Usage**), which means that we have no memory paged out.

Note that the **Mem Usage** column in Task Manager and corresponding performance counters do not count memory that is allocated through AWE. Thus the information is insufficient if AWE is enabled. In this case, you need to look at the memory distribution inside SQL Server to get a full picture.

You can use the **sys.dm\_os\_memory\_clerks** DMV as follows to find out how much memory SQL Server has allocated through AWE mechanism.

select

    sum(awe\_allocated\_kb) / 1024 as [AWE allocated, Mb]

from

    sys.dm\_os\_memory\_clerks

Note that in SQL Server, currently only buffer pool clerks (type = ‘MEMORYCLERK\_SQLBUFFERPOOL’) use this mechanism and only when AWE is enabled.

Relieving external memory pressure by identifying and eliminating major physical memory consumers (if possible) and/or by adding more memory should generally resolve the problems related to memory.

**External virtual memory pressure**

You need to determine if page file(s) have enough space to accommodate current memory allocations. To check this, open Task Manager in Performance view and check the **Commit Charge** section. If **Total** is close to the **Limit**, then there exists the potential that page file space may be running low. **Limit** indicates the maximum amount of memory that can be committed without extending page file space. Note that the **Commit Charge Total** in Task Manager indicates the potential for page file use, not the actual use. Actual use of the page file will increase under physical memory pressure.

Equivalent information can be obtained from the following counters: **Memory: Commit Limit**, **Paging File: %Usage**, **Paging File: %Usage Peak**.

You can roughly estimate the amount of memory that is paged out per process by subtracting the value of **Process: Working Set** from the **Process Private Bytes** counters.

If **Paging File: %Usage Peak** (or Peak Commit Charge) is high, check the System Event Log for events indicating page file growth or notifications of “running low on virtual memory”. You may need to increase the size of your page file(s). **High Paging File: %Usage** indicates a physical memory over commitment and should be considered together with external physical memory pressure (large consumers, adequate amount of RAM installed).

**Internal physical memory pressure**

As internal memory pressure is set by SQL Server itself, a logical step is to look at the memory distribution inside SQL Server by checking for any anomalies in buffer distribution. Normally, the buffer pool accounts for the most of the memory committed by SQL Server. To determine the amount of memory that belongs to the buffer pool, we can take a look at the DBCC MEMORYSTATUS output. In the Buffer Counts section, look for the Target value. The following shows part of DBCC MEMORYSTATUS output after the server has reached its normal load.

Buffer Counts                  Buffers

------------------------------ --------------------

Committed                      201120

Target                         201120

Hashed                         166517

Reserved Potential             143388

Stolen Potential               173556

External Reservation           0

Min Free                       256

Visible                        201120

Available Paging File          460640

Target is computed by SQL Server as the number of 8-KB pages it can commit without causing paging. Target is recomputed periodically and in response to memory low/high notifications from Windows. A decrease in the number of target pages on a normally loaded server may indicate response to an external physical memory pressure.

If SQL Server consumed a lot of memory (as determined by **Process: Private Bytes** or the **Mem Usage** column in Task Manager), see if the Target count amounts for a significant portion of the memory. Note that if AWE is enabled, you have to account for AWE allocated memory either from **sys.dm\_os\_memory\_clerks** or DBCC MEMORYSTATUS output.

Consider the example shown above (AWE not enabled), Target \* 8 KB = 1.53 GB, while the **Process: Private Bytes** for the server is approximately 1.62 GB or the Buffer Pool target accounts for 94% of the memory consumed by SQL Server. Note that if the server is not loaded, Target is likely to exceed the amount reported by **Process: Private Bytes** performance counter, which is normal.

If Target is low, but the server **Process: Private Bytes** or the **Mem Usage** in Task Manager is high, we might be facing internal memory pressure from components that use memory from outside the buffer pool. Components that are loaded into the SQL Server process, such as COM objects, linked servers, extended stored procedures, SQLCLR and others, contribute to memory consumption outside of the buffer pool. There is no easy way to track memory consumed by these components especially if they do not use SQL Server memory interfaces.

Components that are aware of the SQL Server memory management mechanisms use the buffer pool for small memory allocations. If the allocation is bigger than 8 KB, these components use memory outside of the buffer pool through the multi-page allocator interface.

Following is a quick way to check the amount of memory that is consumed through the multi-page allocator.

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

-- amount of mem allocated though multipage

allocator interface select sum(multi\_pages\_kb)

from sys.dm\_os\_memory\_clerks

You can get a more detailed distribution of memory allocated through the multi-page allocator as:

select

    type, sum(multi\_pages\_kb)

from

    sys.dm\_os\_memory\_clerks

where

    multi\_pages\_kb != 0

group by type

type

------------------------------------------ ---------

MEMORYCLERK\_SQLSTORENG                     56

MEMORYCLERK\_SQLOPTIMIZER                   48

MEMORYCLERK\_SQLGENERAL                     2176

MEMORYCLERK\_SQLBUFFERPOOL                  536

MEMORYCLERK\_SOSNODE                        16288

CACHESTORE\_STACKFRAMES                     16

MEMORYCLERK\_SQLSERVICEBROKER               192

MEMORYCLERK\_SNI                            32

If a significant amount of memory is allocated through the multi-page allocator (100-200 MB or more), further investigation is warranted.

If you are seeing large amounts of memory allocated through the multi-page allocator, check the server configuration and try to determine the components that consume most of the memory by using the previous or the following query.

If Target is low but percentage-wise it accounts for most of the memory consumed by SQL Server, look for sources of the external memory pressure as described in the previous subsection ([External Physical Memory Pressure](http://technet.microsoft.com/en-us/library/cc966540.aspx#_External_physical_memory_pressure)) or check the server memory configuration parameters.

If you have the **max server memory** and/or **min server memory** options set, you should compare your target against these values. The **max server memory** option limits the maximum amount of memory consumed by the buffer pool, while the server as a whole can still consume more. The **min server memory** option tells the server not to release buffer pool memory below the setting. If Target is less than the **min server memory** setting and the server is under load, this may indicate that the server operates under the external memory pressure and was never able to acquire the amount specified by this option. It may also indicate the pressure from internal components, as described above. Target count cannot exceed the **max server memory** option setting.

First, check for stolen pages count from DBCC MEMORYSTATUS output.

Buffer Distribution            Buffers

------------------------------ -----------

Stolen                         32871

Free                           17845

Cached                         1513

Database (clean)               148864

Database (dirty)               259

I/O                            0

Latched                        0

A high percentage (>75-80%) of stolen pages relative to target (see the previous fragments of the output) is an indicator of the internal memory pressure.

More detailed information about memory allocation by the server components can be assessed by using the **sys.dm\_os\_memory\_clerks** DMV.

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

-- amount of memory consumed by components

outside the Buffer pool

-- note that we exclude single\_pages\_kb as

they come from BPool

-- BPool is accounted for by the next query

select

    sum(multi\_pages\_kb

        + virtual\_memory\_committed\_kb

        + shared\_memory\_committed\_kb) as

[Overall used w/o BPool, Kb]

from

    sys.dm\_os\_memory\_clerks

where

    type <> 'MEMORYCLERK\_SQLBUFFERPOOL'

-- amount of memory consumed by BPool

-- note that currenlty only BPool uses AWE

select

    sum(multi\_pages\_kb

        + virtual\_memory\_committed\_kb

        + shared\_memory\_committed\_kb

        + awe\_allocated\_kb) as [Used by BPool with AWE, Kb]

from

    sys.dm\_os\_memory\_clerks

where

    type = 'MEMORYCLERK\_SQLBUFFERPOOL'

Detailed information per component can be obtained as follows. (This includes memory allocated from buffer pool as well as outside the buffer pool.)

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

declare @total\_alloc bigint

declare @tab table (

    type nvarchar(128) collate database\_default

    ,allocated bigint

    ,virtual\_res bigint

    ,virtual\_com bigint

    ,awe bigint

    ,shared\_res bigint

    ,shared\_com bigint

    ,topFive nvarchar(128)

    ,grand\_total bigint

);

-- note that this total excludes buffer pool

committed memory as it represents the largest

consumer which is normal

select

    @total\_alloc =

        sum(single\_pages\_kb

            + multi\_pages\_kb

            + (CASE WHEN type <> 'MEMORYCLERK\_SQLBUFFERPOOL'

                THEN virtual\_memory\_committed\_kb

                ELSE 0 END)

            + shared\_memory\_committed\_kb)

from

    sys.dm\_os\_memory\_clerks

print

    'Total allocated (including from Buffer Pool): '

    + CAST(@total\_alloc as varchar(10)) + ' Kb'

insert into @tab

select

    type

    ,sum(single\_pages\_kb + multi\_pages\_kb) as allocated

    ,sum(virtual\_memory\_reserved\_kb) as vertual\_res

    ,sum(virtual\_memory\_committed\_kb) as virtual\_com

    ,sum(awe\_allocated\_kb) as awe

    ,sum(shared\_memory\_reserved\_kb) as shared\_res

    ,sum(shared\_memory\_committed\_kb) as shared\_com

    ,case  when  (

        (sum(single\_pages\_kb

            + multi\_pages\_kb

            + (CASE WHEN type <> 'MEMORYCLERK\_SQLBUFFERPOOL'

                THEN virtual\_memory\_committed\_kb

                ELSE 0 END)

            + shared\_memory\_committed\_kb))/

            (@total\_alloc + 0.0)) >= 0.05

          then type

          else 'Other'

    end as topFive

    ,(sum(single\_pages\_kb

        + multi\_pages\_kb

        + (CASE WHEN type <> 'MEMORYCLERK\_SQLBUFFERPOOL'

            THEN virtual\_memory\_committed\_kb

            ELSE 0 END)

        + shared\_memory\_committed\_kb)) as grand\_total

from

    sys.dm\_os\_memory\_clerks

group by type

order by (sum(single\_pages\_kb + multi\_pages\_kb

+ (CASE WHEN type <>

'MEMORYCLERK\_SQLBUFFERPOOL' THEN

virtual\_memory\_committed\_kb ELSE 0 END) +

shared\_memory\_committed\_kb)) desc

select  \* from @tab

Note that the previous query treats Buffer Pool differently as it provides memory to other components via a single-page allocator. To determine the top ten consumers of the buffer pool pages (via a single-page allocator) you can use the following query.

-- top 10 consumers of memory from BPool

select

    top 10 type,

    sum(single\_pages\_kb) as [SPA Mem, Kb]

from

    sys.dm\_os\_memory\_clerks

group by type

order by sum(single\_pages\_kb) desc

You do not usually have control over memory consumption by internal components. However, determining which components consume most of the memory will help narrow down the investigation of the problem.

**System Monitor (Perfmon)**

You can also check the following counters for signs of memory pressure (see SQL Server Books Online for detailed description):

SQL Server: **Buffer Manager** object

* Low Buffer cache hit ratio
* Low Page life expectancy
* High number of Checkpoint pages/sec
* High number Lazy writes/sec

Insufficient memory and I/O overhead are usually related bottlenecks. See [I/O Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#_I/O_Bottlenecks) in this paper.

**Caches and memory pressure**

An alternative way to look at external and internal memory pressure is to look at the behavior of memory caches.

One of the differences of internal implementation of SQL Server  2005  compared to SQL Server 2000 is uniform caching framework. In order to remove the least recently used entries from caches, the framework implements a clock algorithm. Currently it uses two clock hands—an internal clock hand and an external clock hand.

The internal clock hand controls the size of a cache relative to other caches. It starts moving when the framework predicts that the cache is about to reach its cap.

the external clock hand starts to move when SQL Server as a whole gets into memory pressure. Movement of the external clock hand can be due external as well as internal memory pressure. Do not confuse movement of the internal and external clock hands with internal and external memory pressure.

Information about clock hands movements is exposed through the **sys.dm\_os\_memory\_cache\_clock\_hands** DMV as shown in the following code. Each cache entry has a separate row for the internal and the external clock hand. If you see increasing rounds\_count and removed\_all\_rounds\_count, then the server is under the internal/external memory pressure.

select \*

from

    sys.dm\_os\_memory\_cache\_clock\_hands

where

    rounds\_count > 0

    and removed\_all\_rounds\_count > 0

You can get additional information about the caches such as their size by joining with **sys.dm\_os\_cache\_counters** DMV as follows.

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

select

    distinct cc.cache\_address,

    cc.name,

    cc.type,

    cc.single\_pages\_kb + cc.multi\_pages\_kb as total\_kb,

    cc.single\_pages\_in\_use\_kb + cc.multi\_pages\_in\_use\_kb

    as total\_in\_use\_kb,

    cc.entries\_count,

    cc.entries\_in\_use\_count,

    ch.removed\_all\_rounds\_count,

    ch.removed\_last\_round\_count

from

    sys.dm\_os\_memory\_cache\_counters cc

    join sys.dm\_os\_memory\_cache\_clock\_hands ch on

    (cc.cache\_address =ch.cache\_address)

/\*

--uncomment this block to have the information only

for moving hands caches

where

    ch.rounds\_count > 0

    and ch.removed\_all\_rounds\_count > 0

\*/

order by total\_kb desc

Note that for USERSTORE entries, the amount of pages in use is not reported and thus will be NULL.

**Ring buffers**

Significant amount of diagnostic memory information can be obtained from the **sys.dm\_os\_ring\_buffers** ring buffers DMV. Each ring buffer keeps a record of the last number of notifications of a certain kind. Detailed information on specific ring buffers is provided next.

**RING\_BUFFER\_RESOURCE\_MONITOR**

You can use information from resource monitor notifications to identify memory state changes. Internally, SQL Server has a framework that monitors different memory pressures. When the memory state changes, the resource monitor task generates a notification. This notification is used internally by the components to adjust their memory usage according to the memory state and it is exposed to the user through **sys.dm\_os\_ring\_buffers** DMV as in the following code.

select record

from sys.dm\_os\_ring\_buffers

where ring\_buffer\_type = 'RING\_BUFFER\_RESOURCE\_MONITOR'

A record may look like this:

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

<Record id="1701" type="RING\_BUFFER\_RESOURCE\_MONITOR"

time="149740267">

    <ResourceMonitor>

        <Notification>RESOURCE\_MEMPHYSICAL\_LOW<

        /Notification>

        <Indicators>2</Indicators>

        <NodeId>0</NodeId>

    </ResourceMonitor>

    <MemoryNode id="0">

        <ReservedMemory>1646380</ReservedMemory>

        <CommittedMemory>432388</CommittedMemory>

        <SharedMemory>0</SharedMemory>

        <AWEMemory>0</AWEMemory>

        <SinglePagesMemory>26592</SinglePagesMemory>

        <MultiplePagesMemory>17128</MultiplePagesMemory>

        <CachedMemory>17624</CachedMemory>

    </MemoryNode>

    <MemoryRecord>

        <MemoryUtilization>50</MemoryUtilization>

        <TotalPhysicalMemory>3833132</TotalPhysicalMemory>

        <AvailablePhysicalMemory>3240228<

        /AvailablePhysicalMemory>

        <TotalPageFile>5732340</TotalPageFile>

        <AvailablePageFile>5057100</AvailablePageFile>

        <TotalVirtualAddressSpace>2097024<

        /TotalVirtualAddressSpace>

        <AvailableVirtualAddressSpace>336760

</AvailableVirtualAddressSpace>

        <AvailableExtendedVirtualAddressSpace>0

           </AvailableExtendedVirtualAddressSpace>

    </MemoryRecord>

</Record>

From this record, you can deduce that the server received a low physical memory notification. You can also see the amounts of memory in kilobytes. You can query this information by using the XML capabilities of SQL Server, for example in the following code.

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

select

    x.value('(//Notification)[1]', 'varchar(max)') as [Type],

    x.value('(//Record/@time)[1]', 'bigint') as [Time Stamp],

    x.value('(//AvailablePhysicalMemory)[1]', 'int')

    as [Avail Phys Mem, Kb],

    x.value('(//AvailableVirtualAddressSpace)[1]', 'int')

    as [Avail VAS, Kb]

from

    (select cast(record as xml)

     from sys.dm\_os\_ring\_buffers

     where ring\_buffer\_type = 'RING\_BUFFER\_RESOURCE\_MONITOR')

     as R(x) order by

    [Time Stamp] desc

Upon receiving a memory low notification, the buffer pool recalculates its target. Note that the target count stays within the limits specified by the **min server memory** and **max server memory** options. If the new committed target for the buffer pool is lower than the currently committed buffers, the buffer pool starts shrinking until external physical memory pressure is removed. Note that SQL Server 2000 did not react to physical memory pressure when running with AWE enabled.

**RING\_BUFFER\_OOM**

This ring buffer will contain records indicating server out-of-memory conditions as in the following code example.

select record

from sys.dm\_os\_ring\_buffers

where ring\_buffer\_type = 'RING\_BUFFER\_OOM'

A record may look like this:

<Record id="7301" type="RING\_BUFFER\_OOM" time="345640123">

    <OOM>

        <Action>FAIL\_VIRTUAL\_COMMIT</Action>

        <Resources>4096</Resources>

    </OOM>

This record tells which operation has failed (commit, reserve, or page allocation) and the amount of memory requested.

**RING\_BUFFER\_MEMORY\_BROKER and Internal Memory Pressure**

As internal memory pressure is detected, low memory notification is turned on for components that use the buffer pool as the source of memory allocations. Turning on low memory notification allows reclaiming the pages from caches and other components using them.

Internal memory pressure can also be triggered by adjusting the **max server memory** option or when the percentage of the stolen pages from the buffer pool exceeds 80%.

Internal memory pressure notifications (‘Shrink’) can be observed by querying memory broker ring buffer as in the following code example.

select

    x.value('(//Record/@time)[1]', 'bigint') as [Time Stamp],

    x.value('(//Notification)[1]', 'varchar(100)')

    as [Last Notification]

from

    (select cast(record as xml)

     from sys.dm\_os\_ring\_buffers

     where ring\_buffer\_type = 'RING\_BUFFER\_MEMORY\_BROKER')

     as R(x)

order by

    [Time Stamp] desc

**RING\_BUFFER\_BUFFER\_POOL**

This ring buffer will contain records indicating severe buffer pool failures, including buffer pool out of memory conditions.

select record

from sys.dm\_os\_ring\_buffers

where ring\_buffer\_type = 'RING\_BUFFER\_BUFFER\_POOL'

A record may look like this:

<Record id="1234" type="RING\_BUFFER\_BUFFER\_POOL"

time="345640123">

    < BufferPoolFailure id="FAIL\_OOM">

        <CommittedCount>84344 </CommittedCount>

        <CommittedTarget>84350 </CommittedTarget >

        <FreeCount>20</FreeCount>

        <HashedCount>20345</HashedCount>

        <StolenCount>64001 </StolenCount>

    <ReservedCount>64001 </ReservedCount>

    </ BufferPoolFailure >

This record will tell what failure (FAIL\_OOM, FAIL\_MAP, FAIL\_RESERVE\_ADJUST, FAIL\_LAZYWRITER\_NO\_BUFFERS) and the buffer pool status at the time.

**Internal virtual memory pressure**

VAS consumption can be tracked by using the **sys.dm\_os\_virtual\_address\_dump** DMV. VAS summary can be queries using the following view.

**Note:** Some parts of the code snippet presented in the following table have been displayed in multiple lines only for better readability. These should be entered in a single line.

-- virtual address space summary view

-- generates a list of SQL Server regions

-- showing number of reserved and free regions

of a given size

CREATE VIEW VASummary AS

SELECT

    Size = VaDump.Size,

    Reserved =  SUM(CASE(CONVERT(INT, VaDump.Base)^0)

    WHEN 0 THEN 0 ELSE 1 END),

    Free = SUM(CASE(CONVERT(INT, VaDump.Base)^0)

    WHEN 0 THEN 1 ELSE 0 END)

FROM

(

    --- combine all allocation according with allocation

base, don't take into

    --- account allocations with zero allocation\_base

    SELECT

        CONVERT(VARBINARY, SUM(region\_size\_in\_bytes))

        AS Size,

        region\_allocation\_base\_address AS Base

    FROM sys.dm\_os\_virtual\_address\_dump

    WHERE region\_allocation\_base\_address <> 0x0

    GROUP BY region\_allocation\_base\_address

UNION

       --- we shouldn't be grouping allocations with

       zero allocation base

       --- just get them as is

    SELECT CONVERT(VARBINARY, region\_size\_in\_bytes),

region\_allocation\_base\_address

    FROM sys.dm\_os\_virtual\_address\_dump

    WHERE region\_allocation\_base\_address  = 0x0

)

AS VaDump

GROUP BY Size

The following queries can be used to assess VAS state.

-- available memory in all free regions

SELECT SUM(Size\*Free)/1024 AS [Total avail mem, KB]

FROM VASummary

WHERE Free <> 0

-- get size of largest availble region

SELECT CAST(MAX(Size) AS INT)/1024 AS [Max free size, KB]

FROM VASummary

WHERE Free <> 0

If the largest available region is smaller than 4 MB, we are likely to be experiencing VAS pressure. SQL Server 2005 monitors and responds to VAS pressure. SQL Server 2000 does not actively monitor for VAS pressure, but reacts by clearing caches when an out-of-virtual-memory error occurs.

**General troubleshooting steps in case of memory errors**

The following list outlines general steps that will help you troubleshoot memory errors.

1. Verify if the server is operating under external memory pressure. If external pressure is present, try resolving it first, and then see if the problem/errors still exist.
2. Start collecting performance monitor counters for SQL Server: Buffer Manager, SQL Server: Memory Manager.
3. Verify the memory configuration parameters (**sp\_configure**), **min memory per query**, **min/max server memory**, **awe enabled**, and the **Lock Pages in Memory** privilege. Watch for unusual settings. Correct them as necessary. Account for increased memory requirements for SQL Server 2005.
4. Check for any nondefault **sp\_configure** parameters that might indirectly affect the server.
5. Check for internal memory pressures.
6. Observe DBCC MEMORYSTATUS output and the way it changes when you see memory error messages.
7. Check the workload (number of concurrent sessions, currently executing queries).

**Memory errors**

**701 - There is insufficient system memory to run this query.**

**Causes**

This is very generic out-of-memory error for the server. It indicates a failed memory allocation. It can be due to a variety of reasons, including hitting memory limits on the current workload. With increased memory requirements for SQL Server 2005 and certain configuration settings (such as the **max server memory** option) users are more likely to see this error as compared to SQL Server 2000. Usually the transaction that failed is not the cause of this error.

**Troubleshooting**

Regardless of whether the error is consistent and repeatable (same state) or random (appears at random times with different states), you will need to investigate server memory distribution during the time you see this error. When this error is present, it is possible that the diagnostic queries will fail. Start investigation from external assessment. Follow the steps outlined in [General troubleshooting steps in case of memory errors](http://technet.microsoft.com/en-us/library/cc966540.aspx#_General_troubleshooting_steps_in ca).

Possible solutions include: Remove external memory pressure. Increase the **max server memory** setting. Free caches by using one of the following commands: DBCC FREESYSTEMCACHE, DBCC FREESESSIONCACHE, or DBCC FREEPROCCACHE. If the problem reappears, reduce workload.

**802 - There is insufficient memory available in the buffer pool.**

**Causes**

This error does not necessarily indicate an out-of-memory condition. It might indicate that the buffer pool memory is used by someone else. In SQL Server 2005, this error should be relatively rare.

**Troubleshooting**

Use the general troubleshooting steps and recommendations outlined for the 701 error.

**8628 - A time out occurred while waiting to optimize the query. Rerun the query.**

**Causes**

This error indicates that a query compilation process failed because it was unable to obtain the amount of memory required to complete the process. As a query undergoes through the compilation process, which includes parsing, algebraization, and optimization, its memory requirements may increase. Thus the query will compete for memory resources with other queries. If the query exceeds a predefined timeout (which increases as the memory consumption for the query increases) while waiting for resources, this error is returned. The most likely reason for this is the presence of a number of large query compilations on the server.

**Troubleshooting**

1. Follow general troubleshooting steps to see if the server memory consumption is affected in general.
2. Check the workload. Verify the amounts of memory consumed by different components. (See [Internal Physical Memory Pressure](http://technet.microsoft.com/en-us/library/cc966540.aspx#_Internal_physical_memory_pressure) earlier in this paper.)
3. Check the output of DBCC MEMORYSTATUS for the number of waiters at each gateway (this information will tell you if there are other queries running that consume significant amounts of memory).
4. Small Gateway                  Value
5. ------------------------------ --------------------
6. Configured Units               8
7. Available Units                8
8. Acquires                       0
9. Waiters                        0
10. Threshold Factor               250000
11. Threshold                      250000
13. (6 row(s) affected)
15. Medium Gateway                 Value
16. ------------------------------ --------------------
17. Configured Units               2
18. Available Units                2
19. Acquires                       0
20. Waiters                        0
21. Threshold Factor               12
23. (5 row(s) affected)
25. Big Gateway                    Value
26. ------------------------------ --------------------
27. Configured Units               1
28. Available Units                1
29. Acquires                       0
30. Waiters                        0
31. Threshold Factor               8
32. Reduce workload if possible.

**8645 - A time out occurred while waiting for memory resources to execute the query. Rerun the query.**

**Causes**

This error indicates that many concurrent memory intensive queries are being executed on the server. Queries that use sorts (ORDER BY) and joins may consume significant amount of memory during execution. Query memory requirements are significantly increased if there is a high degree of parallelism enabled or if a query operates on a partitioned table with non-aligned indexes. A query that cannot get the memory resources it requires within the predefined timeout (by default, the timeout is 25 times the estimated query cost or the **sp\_configure** ‘query wait’ amount if set) receives this error. Usually, the query that receives the error is not the one that is consuming the memory.

**Troubleshooting**

1. Follow general steps to assess server memory condition.
2. Identify problematic queries: verify if there is a significant number of queries that operate on partitioned tables, check if they use non-aligned indexes, check if there are many queries involving joins and/or sorts.
3. Check the **sp\_configure** parameters **degree of parallelism** and **min memory per query**. Try reducing the degree of parallelism and verify if **min memory per query** is not set to a high value. If it is set to a high value, even small queries will acquire the specified amount of memory.
4. To find out if queries are waiting on RESOURCE\_SEMAPHORE, see [Blocking](http://technet.microsoft.com/en-us/library/cc966540.aspx#_Blocking) later in this paper.

**8651 - Could not perform the requested operation because the minimum query memory is not available. Decrease the configured value for the 'min memory per query' server configuration option.**

**Causes**

Causes in part are similar to the 8645 error; it may also be an indication of general memory low condition on the server. A **min memory per query** option setting that is too high may also generate this error.

**Troubleshooting**

1. Follow general memory error troubleshooting steps.
2. Verify the **sp\_configure min memory per query** option setting.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**I/O Bottlenecks**

SQL Server performance depends heavily on the I/O subsystem. Unless your database fits into physical memory, SQL Server constantly brings database pages in and out of the buffer pool. This generates substantial I/O traffic. Similarly, the log records need to be flushed to the disk before a transaction can be declared committed. And finally, SQL Server uses **tempdb** for various purposes such as to store intermediate results, to sort, to keep row versions and so on. So a good I/O subsystem is critical to the performance of SQL Server.

Access to log files is sequential except when a transaction needs to be rolled back while access to data files, including **tempdb**, is randomly accessed. So as a general rule, you should have log files on a separate physical disk than data files for better performance. The focus of this paper is not how to configure your I/O devices but to describe ways to identify if you have I/O bottleneck. Once an I/O bottleneck is identified, you may need to reconfigure your I/O subsystem.

If you have a slow I/O subsystem, your users may experience performance problems such as slow response times, and tasks that abort due to timeouts.

You can use the following performance counters to identify I/O bottlenecks. Note, these AVG values tend to be skewed (to the low side) if you have an infrequent collection interval. For example, it is hard to tell the nature of an I/O spike with 60-second snapshots. Also, you should not rely on one counter to determine a bottleneck; look for multiple counters to cross check the validity of your findings.

* **PhysicalDisk Object: Avg. Disk Queue Length** represents the average number of physical read and write requests that were queued on the selected physical disk during the sampling period. If your I/O system is overloaded, more read/write operations will be waiting. If your disk queue length frequently exceeds a value of 2 during peak usage of SQL Server, then you might have an I/O bottleneck.
* **Avg. Disk Sec/Read** is the average time, in seconds, of a read of data from the disk. Any number
* Less than 10 ms - very good
* Between 10 - 20 ms - okay
* Between 20 - 50 ms - slow, needs attention
* Greater than 50 ms – Serious I/O bottleneck
* **Avg. Disk Sec/Write** is the average time, in seconds, of a write of data to the disk. Please refer to the guideline in the previous bullet.
* **Physical Disk: %Disk Time** is 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 is greater than 50 percent, it represents an I/O bottleneck.
* **Avg. Disk Reads/Sec** is the rate of read operations on the disk. You need to make sure that this number is less than 85 percent of the disk capacity. The disk access time increases exponentially beyond 85 percent capacity.
* **Avg. Disk Writes/Sec** is the rate of write operations on the disk. Make sure that this number is less than 85 percent of the disk capacity. The disk access time increases exponentially beyond 85 percent capacity.

When using above counters, you may need to adjust the values for RAID configurations using the following formulas.

Raid 0 -- I/Os per disk = (reads + writes) / number of disks

Raid 1 -- I/Os per disk = [reads + (2 \* writes)] / 2

Raid 5 -- I/Os per disk = [reads + (4 \* writes)] / number of disks

Raid 10 -- I/Os per disk = [reads + (2 \* writes)] / number of disks

For example, you have a RAID-1 system with two physical disks with the following values of the counters.

Disk Reads/sec            80

Disk Writes/sec           70

Avg. Disk Queue Length    5

In that case, you are encountering (80 + (2 \* 70))/2 = 110 I/Os per disk and your disk queue length = 5/2 = 2.5 which indicates a border line I/O bottleneck.

You can also identify I/O bottlenecks by examining the latch waits. These latch waits account for the physical I/O waits when a page is accessed for reading or writing and the page is not available in the buffer pool. When the page is not found in the buffer pool, an asynchronous I/O is posted and then the status of the I/O is checked. If I/O has already completed, the worker proceeds normally. Otherwise, it waits on PAGEIOLATCH\_EX or PAGEIOLATCH\_SH, depending upon the type of request. The following DMV query can be used to find I/O latch wait statistics.

Select  wait\_type,

        waiting\_tasks\_count,

        wait\_time\_ms

from    sys.dm\_os\_wait\_stats

where    wait\_type like 'PAGEIOLATCH%'

order by wait\_type

wait\_type       waiting\_tasks\_count  wait\_time\_ms   signal\_wait\_time\_ms

-----------------------------------------------------------------------

PAGEIOLATCH\_DT  0                    0                    0

PAGEIOLATCH\_EX  1230                 791                  11

PAGEIOLATCH\_KP  0                    0                    0

PAGEIOLATCH\_NL  0                    0                    0

PAGEIOLATCH\_SH  13756                7241                 180

PAGEIOLATCH\_UP  80                   66                   0

Here the latch waits of interest are the underlined ones. When the I/O completes, the worker is placed in the runnable queue. The time between I/O completions until the time the worker is actually scheduled is accounted under the signal\_wait\_time\_ms column. You can identify an I/O problem if your waiting\_task\_counts and wait\_time\_ms deviate significantly from what you see normally. For this, it is important to get a baseline of performance counters and key DMV query outputs when SQL Server is running smoothly. These wait\_types can indicate whether your I/O subsystem is experiencing a bottleneck, but they do not provide any visibility on the physical disk(s) that are experiencing the problem.

You can use the following DMV query to find currently pending I/O requests. You can execute this query periodically to check the health of I/O subsystem and to isolate physical disk(s) that are involved in the I/O bottlenecks.

select

    database\_id,

    file\_id,

    io\_stall,

    io\_pending\_ms\_ticks,

    scheduler\_address

from    sys.dm\_io\_virtual\_file\_stats(NULL, NULL)t1,

        sys.dm\_io\_pending\_io\_requests as t2

where    t1.file\_handle = t2.io\_handle

A sample output is as follows. It shows that on a given database, there are three pending I/Os at this moment. You can use the database\_id and file\_id to find the physical disk the files are mapped to. The io\_pending\_ms\_ticks represent the total time individual I/Os are waiting in the pending queue.

Database\_id File\_Id io\_stall io\_pending\_ms\_ticks scheduler\_address

----------------------------------------------------------------------

6           1        10804        78            0x0227A040

6           1        10804        78            0x0227A040

6           2        101451       31            0x02720040

**Resolution**

When you have identified an I/O bottleneck, you can address it by doing one or more of the following:

* Check the memory configuration of SQL Server. If SQL Server has been configured with insufficient memory, it will incur more I/O overhead. You can examine following counters to identify memory pressure
  + Buffer Cache hit ratio
  + Page Life Expectancy
  + Checkpoint pages/sec
  + Lazywrites/sec

For more information on the memory pressure, see [Memory Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#_Memory_Bottlenecks).

* Increase I/O bandwidth.
  + Add more physical drives to the current disk arrays and/or replace your current disks with faster drives. This helps to boost both read and write access times. But don't add more drives to the array than your I/O controller can support.
  + Add faster or additional I/O controllers. Consider adding more cache (if possible) to your current controllers.
* Examine execution plans and see which plans lead to more I/O being consume. It is possible that a better plan (for example, index) can minimize I/O. If there are missing indexes, you may want to run Database Engine Tuning Advisor to find missing indexes

The following DMV query can be used to find which batches/requests are generating the most I/O. You will notice that we are not accounting for physical writes. This is ok if you consider how databases work. The DML/DDL statements within a request do not directly write data pages to disk. Instead, the physical writes of pages to disks is triggered by statements only by committing transactions. Usually physical writes are done by either by Checkpoint or by the SQL Server lazy writer. A DMV query like the following can be used to find the top five requests that generate the most I/Os. Tuning those queries so that they perform fewer logical reads can relieve pressure on the buffer pool. This allows other requests to find the necessary data in the buffer pool in repeated executions (instead of performing physical I/O). Hence, overall system performance is improved.

select top 5

    (total\_logical\_reads/execution\_count) as avg\_logical\_reads,

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

    (total\_physical\_reads/execution\_count) as avg\_phys\_reads,

     Execution\_count,

    statement\_start\_offset as stmt\_start\_offset,

    sql\_handle,

    plan\_handle

from sys.dm\_exec\_query\_stats

order by

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

You can, of course, change this query to get different views on the data. For example, to generate the top five requests that generate most I/Os in single execution, you can order by:

    (total\_logical\_reads + total\_logical\_writes)/execution\_count

Alternatively, you may want to order by physical I/Os and so on. However, logical read/write numbers are very helpful in determining whether or not the plan chosen by the query is optimal. For example, it may be doing a table scan instead of using an index. Some queries, such as those that use nested loop joins may have high logical counters but be more cache-friendly since they revisit the same pages.

**Example**: Let us take the following two batches consisting of two SQL queries where each table has 1000 rows and rowsize > 8000 (one row per page).

Batch-1

select

    c1,

    c5

from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4

order by c2

Batch-2

select \* from t1

For the purpose of this example, before running the DMV query, we clear the buffer pool and the procedure cache by running the following commands.

checkpoint

dbcc freeproccache

dbcc dropcleanbuffers

Here is the output of the DMV query. You will notice two rows representing the two batches.

Avg\_logical\_reads Avg\_logical\_writes Avg\_phys\_reads Execution\_count

stmt\_start\_offset

-----------------------------------------------------------------------

---------------

2794                1                385                1

    0

1005                0                0                  1

    146

sql\_handle                                         plan\_handle

-----------------------------------------------------------------------

-----

0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB

x0600050099EC8520A8619803000000000000000000000000

0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB

x0600050099EC8520A8619803000000000000000000000000

You will notice that the second batch only incurs logical reads but no physical I/O. This is because the data it needs was already cached by the first query (assuming there was sufficient memory).

You can get the text of the query by running the following query.

select text

from sys.dm\_exec\_sql\_text(

     0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB)

Here is the output.

select

    c1,

    c5

from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4

order by c2

You can also find out the string for the individual statement by executing the following:

select

    substring(text,

              (<statement\_start\_offset>/2),

              (<statement\_end\_offset> -<statement\_start\_offset>)/2)

from sys.dm\_exec\_sql\_text

(0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB)

The value of statement\_start\_offest and statement\_end\_offset need to be divided by two in order to compensate for the fact that SQL Server stores this kind of data in Unicode. A statement\_end\_offset value of -1, indicates that the statement does go up to the end of the batch. However the **substring()** function does not accommodate -1 as a valid value. Instead of using -1 as (<statement\_end\_offset> -<statement\_start\_offset>)/2, one should enter the value 64000, which should make sure that the statement is covered in all cases. With this method, a long-running or resource-consuming statement can be filtered out of a large stored procedure or batch.

Similarly, you can run the following query to find to the query plan to identify if the large number of I/Os is a result of a poor plan choice.

select \*

from sys.dm\_exec\_query\_plan

    (0x0600050099EC8520A8619803000000000000000000000000)

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966540.aspx#mainSection)

**Tempdb**

**Tempdb** globally stores both internal and user objects and the temporary tables, objects, and stored procedures that are created during SQL Server operation.

There is a single **tempdb** for each SQL Server instance. It can be a performance and disk space bottleneck. The **tempdb** can become overloaded in terms of space available and excessive DDL/DML operations. This can cause unrelated applications running on the server to slow down or fail.

Some of the common issues with **tempdb** are as follows:

* Running out of storage space in **tempdb**.
* Queries that run slowly due to the I/O bottleneck in **tempdb**. This is covered under [I/O Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#_I/O_Bottlenecks).
* Excessive DDL operations leading to a bottleneck in the system tables.
* Allocation contention.

Before we start diagnosing problems with **tempdb**, let us first look at how the space in **tempdb** is used. It can be grouped into four main categories.

|  |  |
| --- | --- |
| User objects | These are explicitly created by user sessions and are tracked in system catalog. They include the following:   * Table and index. * Global temporary table (##t1) and index. * Local temporary table (#t1) and index.   + Session scoped.   + Stored procedure scoped in which it was created. * Table variable (@t1).   + Session scoped.   + Stored procedure scoped in which it was created. |
| Internal objects | These are statement scoped objects that are created and destroyed by SQL Server to process queries. These are not tracked in the system catalog. They include the following:   * Work file (hash join) * Sort run * Work table (cursor, spool and temporary large object data type (LOB) storage)   As an optimization, when a work table is dropped, one IAM page and an extent is saved to be used with a new work table.  There are two exceptions; the temporary LOB storage is batch scoped and cursor worktable is session scoped. |
| Version Store | This is used for storing row versions. MARS, online index, triggers and snapshot-based isolation levels are based on row versioning. This is new in SQL Server 2005. |
| Free Space | This represents the disk space that is available in **tempdb**. |

The total space used by tempdb equal to the User Objects plus the Internal Objects plus the Version Store plus the Free Space.

This free space is same as the performance counter free space in **tempdb**.

**Monitoring tempdb space**

It is better to prevent a problem then work to solve it later. You can use the following performance counters to monitor the amount of space tempdb is using.

* **Free Space in tempdb** (KB). This counter tracks free space in **tempdb** in kilobytes. Administrators can use this counter to determine if **tempdb** is running low on free space.

However, identifying how the different categories, as defined above, are using the disk space in **tempdb** is a more interesting, and productive, question.

The following query returns the **tempdb** space used by user and by internal objects. Currently, it provides information for **tempdb** only.

Select

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

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

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

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

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

Where database\_id = 2

Here is one sample output (with space in KBs).

user\_objets\_kb   internal\_objects\_kb   version\_store\_kb   freespace\_kb

---------------- -------------------- ------------------ ------------

8736               128                    64                    448

Note that these calculations don’t account for pages in mixed extents. The pages in mixed extents can be allocated to user and internal objects.