Troubleshooting Performance Problems in SQL Server 2005

SQL Server Technical Article

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

Published: October 2005

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.

Copyright

Table of Contents

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.

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.

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.

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.

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 [Statistics Used by the Query Optimizer in Microsoft SQL Server 2005](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).

```
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.

```
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**

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://msdn.microsoft.com/library/en-us/memory/base/virtual_address_space.asp) 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://msdn.microsoft.com/library/en-us/memory/base/large_memory_support.asp) 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	2 GB	64 GB	Yes
with /3GB boot parameter[1]	3 GB	16 GB	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

[1] For more information about boot parameters consult SQL Server Books Online "Using AWE" topic.

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 mechanism[2] 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.

[2] For more information, see QueryMemoryResourceNotification on MSDN.

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.

```
-- 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) 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.

```
-- 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.)

```
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 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.

```
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:

```
<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.

```
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.

```
-- 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.

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 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).

```
Small Gateway                  Value
------------------------------ --------------------
Configured Units               8
Available Units                8
Acquires                       0
Waiters                        0
Threshold Factor               250000
Threshold                      250000

(6 row(s) affected)

Medium Gateway                 Value
------------------------------ --------------------
```

```
Configured Units               2
Available Units                2
Acquires                       0
Waiters                        0
Threshold Factor               12

(5 row(s) affected)

Big Gateway                    Value
------------------------------ --------------------
Configured Units               1
Available Units                1
Acquires                       0
Waiters                        0
Threshold Factor               8
```

4. 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

5. Follow general steps to assess server memory condition.
6. 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.
7. 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.
8. To find out if queries are waiting on RESOURCE_SEMAPHORE, see 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.

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.
- 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)
```

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.
- 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.

Troubleshooting space issues

User objects, internal objects, and version storage can all cause space issues in **tempdb**. In this section, we consider how you can troubleshoot each of these categories.

User objects

Since user objects are not owned by any specific sessions, you need to understand the specifications of the application that created them and adjust the **tempdb** size requirements accordingly. You can find the space used by individual user objects by executing `exec sp_spaceused @objname='<user-object>'`. For example, you can run the following script to enumerate all the **tempdb** objects.

```
DECLARE userobj_cursor CURSOR FOR
select
     sys.schemas.name + '.' + sys.objects.name
from sys.objects, sys.schemas
where object_id > 100 and
      type_desc = 'USER_TABLE'and
      sys.objects.schema_id = sys.schemas.schema_id
go

open userobj_cursor
go

declare @name varchar(256)
fetch userobj_cursor into @name
while (@@FETCH_STATUS = 0)
begin
    exec sp_spaceused @objname = @name
        fetch userobj_cursor into @name
end
close userobj_cursor
```

Version store

SQL Server 2005 provides a row versioning framework that is used to implement new and existing features. Currently, the following features use row versioning framework. For more information about the following features, see SQL Server Books Online.

- Triggers
- MARS
- Online index
- Row versioning-based isolation levels: requires setting an option at database level

Row versions are shared across sessions. The creator of the row version has no control over when the row version can be reclaimed. You will need to find and then possibly kill the longest running transaction that is preventing the row version cleanup.

The following query returns the top two longest running transactions that depend on the versions in the version store.

```
select top 2

    transaction_id,

    transaction_sequence_num,

    elapsed_time_seconds

from sys.dm_tran_active_snapshot_database_transactions

order by elapsed_time_seconds DESC
```

Here is a sample output that shows that a transaction with XSN 3 and Transaction ID 8609 has been active for 6523 seconds.

```
transaction_id       transaction_sequence_num elapsed_time_seconds

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

8609                 3                        6523

20156                25                       783
```

Since the second transaction has been active for a relatively short period, you can possibly free up a significant amount of version store by killing the first transaction. However, there is no way to estimate how much version space will be freed up by killing this transaction. You may need to kill few a more transactions to free up significant space.

You can mitigate this problem by either sizing your **tempdb** properly to account for the version store or by eliminating, where possible, long running transactions under snapshot isolation or long running queries under read-committed-snapshot. You can roughly estimate the size of the version store that is needed by using the following formula. (A factor of two is needed to account for the worst-case scenario, which occurs when the two longest running transactions overlap.)

```
[Size of version store] = 2 * [version store data generated per minute] *

[longest running time (minutes) of the transaction]
```

In all databases that are enabled for row versioning based isolation levels, the version store data generated per minute for a transaction is about the same as log generated per minute. However. there are some exceptions: only differences are logged for updates; and a newly inserted data row is not versioned but may be logged depending if it is a bulk-logged operation and the recovery mode is not set to full recovery.

You can also use the **Version Generation Rate** and **Version Cleanup Rate** performance counters to fine tune your computation. If your **Version Cleanup Rate** is 0, this implies that there is a long running transaction that is preventing the version store cleanup.

Incidentally, before generating an out of **tempdb** space error, SQL Server 2005 makes a last ditch attempt by forcing the version stores to shrink. During the shrink process, the longest running transactions that have not yet generated any row versions are marked as victims. This frees up the version space used by them. Message 3967 is generated in the error log for each such victim transaction. If a transaction is marked as a victim, it can no longer read the row versions in the version store or create new ones. Message 3966 is generated and the transaction is rolled back when the victim transaction attempts to read row versions. If the shrink of the version store succeeds, then more space is available in **tempdb**. Otherwise, **tempdb** runs out of space.

Internal Objects

Internal objects are created and destroyed for each statement, with exceptions as outlined in the previous table. If you notice that a huge amount of **tempdb** space is allocated, you will need to know which session or tasks are consuming the space and then possibly take the corrective action.

SQL Server 2005 provides two additional DMVs: **sys.dm_db_session_space_usage** and **sys.dm_db_task_space_usage** to track **tempdb** space that is allocated to sessions and tasks respectively. Though tasks are run in the context of sessions, the space used by tasks is accounted for under sessions only after the task complete.

You can use the following query to find the top sessions that are allocating internal objects. Note that this query includes only the tasks that have been completed in the sessions.

```
select
    session_id,
    internal_objects_alloc_page_count,
    internal_objects_dealloc_page_count
from sys.dm_db_session_space_usage
order by internal_objects_alloc_page_count DESC
```

You can use the following query to find the top user sessions that are allocating internal objects, including currently active tasks.

```
SELECT
    t1.session_id,
    (t1.internal_objects_alloc_page_count + task_alloc) as allocated,
    (t1.internal_objects_dealloc_page_count + task_dealloc) as
    deallocated
from sys.dm_db_session_space_usage as t1,
    (select session_id,
        sum(internal_objects_alloc_page_count)
            as task_alloc,
    sum (internal_objects_dealloc_page_count) as
        task_dealloc
      from sys.dm_db_task_space_usage group by session_id) as t2
where t1.session_id = t2.session_id and t1.session_id >50
order by allocated DESC
```

Here is a sample output.

```
session_id allocated            deallocated
---------- -------------------- --------------------
52         5120                 5136
51         16                   0
```

Once you have isolated the task or tasks that are generating a lot of internal object allocations, you can find out which Transact-SQL statement it is and its query plan for a more detailed analysis.

```
select
    t1.session_id,
    t1.request_id,
    t1.task_alloc,
    t1.task_dealloc,
    t2.sql_handle,
    t2.statement_start_offset,
    t2.statement_end_offset,
    t2.plan_handle
from (Select session_id,
             request_id,
             sum(internal_objects_alloc_page_count) as task_alloc,
             sum (internal_objects_dealloc_page_count) as task_dealloc
      from sys.dm_db_task_space_usage
      group by session_id, request_id) as t1,
      sys.dm_exec_requests as t2
where t1.session_id = t2.session_id and
     (t1.request_id = t2.request_id)
order by t1.task_alloc DESC
```

Here is a sample output.

```
session_id request_id  task_alloc           task_dealloc
---------------------------------------------------------
52         0           1024                 1024
sql_handle                                      statement_start_offset
-----------------------------------------------------------------------
0x02000000D490961BDD2A8BE3B0FB81ED67655EFEEB360172   356

statement_end_offset  plan_handle
---------------------------------
-1                    0x06000500D490961BA8C19503000000000000000000000000
```

You can use the `sql_handle` and `plan_handle` to get the SQL statement and the query plan as follows:

```
select text from sys.dm_exec_sql_text(@sql_handle)
select * from sys.dm_exec_query_plan(@plan_handle)
```

Note that it is possible that a query plan may not be in the cache when you want to access it. To guarantee the availability of the query plans, you will need to poll the plan cache frequently and save the results, preferably in a table, so that it can be queried later.

When SQL Server is restarted, the **tempdb** size goes back to the initially configured size and it grows based on the requirements. This can lead to fragmentation of the **tempdb** and can incur an overhead, including the blocking of the allocation of new extents during the database auto-grow, and expanding the size of the **tempdb**. This can impact the performance of your workload. It is recommended that you pre-allocate **tempdb** to the appropriate size.

Excessive DDL and allocation operations

Two sources of contention in **tempdb** can result in the following situations.

- Creating and dropping large number of temporary tables and table variables can cause contention on metadata. In SQL Server 2005, local temporary tables and table variables are cached to minimize metadata contention. However, the following conditions must be satisfied, otherwise the temp objects are not cached.
  - Named constraints are not created.
  - Data Definition Language (DDL) statements that affect the table are not run after the temp table has been created, such as the CREATE INDEX or CREATE STATISTICS statements.
  - Temp object is not created by using dynamic SQL, such as: sp_executesql N'create table #t(a int)'.
  - Temp object is created inside another object, such as a stored procedure, trigger, and user-defined function; or is the return table of a user-defined, table-valued function.
- Typically, most temporary/work tables are heaps; therefore, an insert, delete, or drop operation can cause heavy contention on Page Free Space (PFS) pages. If most of these tables are under 64 KB and use mixed extent for allocation or deal location, this can put heavy contention on Shared Global Allocation Map (SGAM) pages. SQL Server 2005 caches one data page and one IAM page for local temporary tables to minimize allocation contention. Worktable caching is improved. When a query execution plan is cached, the work tables needed by the plan are not dropped across multiple executions of the plan but merely truncated. In addition, the first nine pages for the work table are kept.

Since SGAM and PFS pages occur at fixed intervals in data files, it is easy to find their resource description. So, for example, 2:1:1 represents the first PFS page in the **tempdb** (database-id = 2, file-id =1, page-id = 1) and 2:1:3 represents the first SGAM page. SGAM pages occur after every 511232 pages and each PFS page occurs after every 8088 pages. You can use this to find all other PFS and SGAM pages across all files in **tempdb**. Any time a task is waiting to acquire latch on these pages, it will show up in **sys.dm_os_waiting_tasks**. Since latch waits are transient, you will need to query this table frequently (about once every 10 seconds) and collect this data for analysis later. For example, you can use the following query to load all tasks waiting on **tempdb** pages into a waiting_tasks table in the analysis database.

```
-- get the current timestamp
declare @now datetime
select @now = getdate()

-- insert data into a table for later analysis
insert into analysis..waiting_tasks
      select
          session_id,
          wait_duration_ms,
```

```
            resource_description,
            @now
        from sys.dm_os_waiting_tasks
        where wait_type like 'PAGE%LATCH_%' and
              resource_description like '2:%'
```

Any time you see tasks waiting to acquire latches on **tempdb** pages, you can analyze to see if it is due to PFS or SGAM pages. If it is, this implies allocation contention in **tempdb**. If you see contention on other pages in **tempdb**, and if you can identify that a page belongs to the system table, this implies contention due to excessive DDL operations.

You can also monitor the following Perfmon counters for any unusual increase in the temporary objects allocation/deal location activity.

- SQL Server:**Access Methods\Workfiles Created /Sec**
- SQL Server:**Access Methods\Worktables Created /Sec**
- SQL Server:**Access Methods\Mixed Page Allocations /Sec**
- SQL Server:**General Statistics\Temp Tables Created /Sec**
- SQL Server:**General Statistics\Temp Tables for destruction**

Resolution

If the contention in **tempdb** is due to excessive DDL operation, you will need to look at your application and see if you can minimize the DDL operation. You can try the following suggestions.

- Starting with SQL Server 2005, the temporary objects are cached under conditions as described earlier. However, if you are still encountering significant DDL contention, you need to look at what temporary objects are not being cached and where do they occur. If such objects occur inside a loop or a stored procedure, consider moving them out of the loop or the stored procedure.
- Look at query plans to see if some plans create lot of temporary objects, spools, sorts, or worktables. You may need to eliminate some temporary objects. For example, creating an index on a column that is used in ORDER BY may eliminate the sort.

If the contention is due to the contention in SGAM and PFS pages, you can mitigate it by trying the following:

- Increase the **tempdb** data files by an equal amount to distribute the workload across all of the disks and files. Ideally, you want to have as many files as there are CPUs (taking into account the affinity).
- Use TF-1118 to eliminate mixed extent allocations.

Slow-Running Queries

Slow or long running queries can contribute to excessive resource consumption and be the consequence of blocked queries.

Excessive resource consumption is not restricted to CPU resources, but can also include I/O storage bandwidth and memory bandwidth. Even if SQL Server queries are designed to avoid full table scans by restricting the result set with a reasonable WHERE clause, they might not perform as expected if there is not an appropriate index supporting that particular query. Also, WHERE clauses can be dynamically constructed by applications, depending on the user input. Given this, existing indexes cannot cover all possible cases of restrictions. Excessive CPU, I/O, and memory consumption by Transact-SQL statements are covered earlier in this white paper.

In addition to missing indexes, there may be indexes that are not used. As all indexes have to be maintained, this does not impact the performance of a query, but does impact the DML queries.

Queries can also run slowly because of wait states for logical locks and for system resources that are blocking the query. The cause of the blocking can be a poor application design, bad query plans, the lack of useful indexes, and an SQL Server instance that is improperly configured for the workload.

This section focuses on two causes of a slow running query—blocking and index problems.

Blocking

Blocking is primarily waits for logical locks, such as the wait to acquire an X lock on a resource or the waits that results from lower level synchronization primitives such as latches.

Logical lock waits occur when a request to acquire a non-compatible lock on an already locked resource is made. While this is needed to provide the data consistency based on the transaction isolation level at which a particular Transact-SQL statement is running, it does give the end user a perception that SQL Server is running slowly. When a query is blocked, it is not consuming any system resources so you will find it is taking longer but the resource consumption is low. For details on the concurrency control and blocking, see SQL Server Books Online.

Waits on lower level synchronization primitives can result if your system is not configured to handle the workload.

The common scenarios for blocking/waits are:

- Identifying the blocker
- Identifying long blocks
- Blocking per object
- Page latching issues
- Overall performance effect of blocking using SQL Server waits

A SQL Server session is placed in a wait state if system resources (or locks) are not currently available to service the request. In other words, the resource has a queue of outstanding requests. DMVs can provide information for any sessions that are waiting on resources.

SQL Server 2005 provides more detailed and consistent wait information, reporting approximately 125 wait types compared with the 76 wait types available in SQL Server 2000. The DMVs that provide this information range from **sys.dm_os_wait_statistics** for overall and cumulative waits for SQL Server to the session-specific **sys.dm_os_waiting_tasks** that breaks down waits by session. The following DMV provides details on the wait queue of the tasks that are waiting on some resource. It is a simultaneous representation of all wait queues in the system. For example, you can find out the details about the blocked session 56 by running the following query.

```
select * from sys.dm_os_waiting_tasks where session_id=56

waiting_task_address session_id exec_context_id wait_duration_ms
   wait_type
 resource_address   blocking_task_address blocking_session_id
blocking_exec_context_id resource_description
-------------------- ---------- --------------- -------------------- -----
------------------------------------------------------- ------------------
--------------------- ------------------- ------------------------ -------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
---------------------------
0x022A8898           56         0               1103500
LCK_M_S                                                             0x03696820
     0x022A8D48            56                  NULL
ridlock fileid=1 pageid=143 dbid=9 id=lock3667d00 mode=X
associatedObjectId=72057594038321152
```

This result shows that session 56 is blocked by session 53 and that session 56 has been waiting for a lock for 1103500 milliseconds.

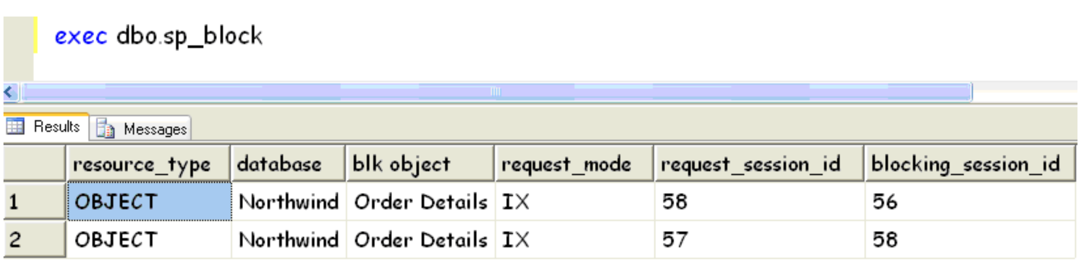
To find sessions that have been granted locks or waiting for locks, you can use the **sys.dm_tran_locks** DMV*.* Each row represents a currently active request to the lock manager that has either been granted or is waiting to be granted as the request is blocked by a request that has already been granted. For regular locks, a granted request indicates that a lock has been granted on a resource to the requestor. A waiting request indicates that the request has not yet been granted. For example, the following query shows that session 56 is blocked on the resource 1:143:3 that is held in X mode by session 53.

```
select
    request_session_id as spid,
    resource_type as rt,
    resource_database_id as rdb,
    (case resource_type
      WHEN 'OBJECT' then object_name(resource_associated_entity_id)
      WHEN 'DATABASE' then ' '
      ELSE (select object_name(object_id)
```

```
            from sys.partitions
            where hobt_id=resource_associated_entity_id)
    END) as objname,
    resource_description as rd,
    request_mode as rm,
    request_status as rs
from sys.dm_tran_locks

Here is the sample output
spid     rt           rdb         objname       rd            rm
rs
---------------------------------------------------------------------------
---
56     DATABASE     9                               S           GRANT
53     DATABASE     9                               S           GRANT
56     PAGE         9        t_lock      1:143      IS          GRANT
53     PAGE         9        t_lock      1:143      IX          GRANT
53     PAGE         9        t_lock      1:153      IX          GRANT
56    OBJECT        9        t_lock                 IS          GRANT
53    OBJECT        9         t_lock                IX          GRANT
53     KEY          9         t_lock      (a400c34cb X          GRANT
53     RID          9         t_lock      1:143:3    X          GRANT
56     RID          9         t_lock      1:143:3    S         WAIT
```

You can, in fact, join the above two DMVs as shown with stored proc **sp_block**. The block report in Figure 1 lists sessions that are blocked and the sessions that are blocking them. You can find the source code in Appendix B. You can alter the stored procedure, if needed, to add/remove the attributes in the Select List. The optional @spid parameter provides details on the lock request and the session that is blocking this particular spid.



	resource_type	database	blk object	request_mode	request_session_id	blocking_session_id
1	OBJECT	Northwind	Order Details	IX	58	56
2	OBJECT	Northwind	Order Details	IX	57	58

Figure 1: sp_block report

In SQL Server 2000, you can see what spids are blocked information by using the following statement.

```
select * from master..sysprocesses where blocked <> 0.
```

The associated locks can be seen with the stored procedure **sp_lock**.

Identifying long blocks

As mentioned earlier, blocking in SQL Server is common and is a manifestation of the logical locks that are needed to maintain the transactional consistency. However, when the wait for locks exceed a threshold, it may impact the response time. To identify long running blocking, you can use BlockedProcessThreshold configuration parameter to establish a user configured server-wide block threshold. The threshold defines a duration in seconds. Any block that exceeds this threshold will fire an event that can be captured by SQL Trace.

For example, a 200-second blocked process threshold can be configured in SQL Server Management Studio as follows:

1. Execute Sp_configure 'blocked process threshold', 200
2. Reconfigure with override

Once the blocked process threshold is established, the next step is to capture the trace event. The trace events of blocking events that exceed the user configured threshold can be captured with SQL Trace or Profiler.

3. If using SQL Trace, use sp_trace_setevent and event_id=137.
4. If using SQL Server Profiler, select the Blocked Process Report event class (under the Errors and Warnings object). See Figure 2.

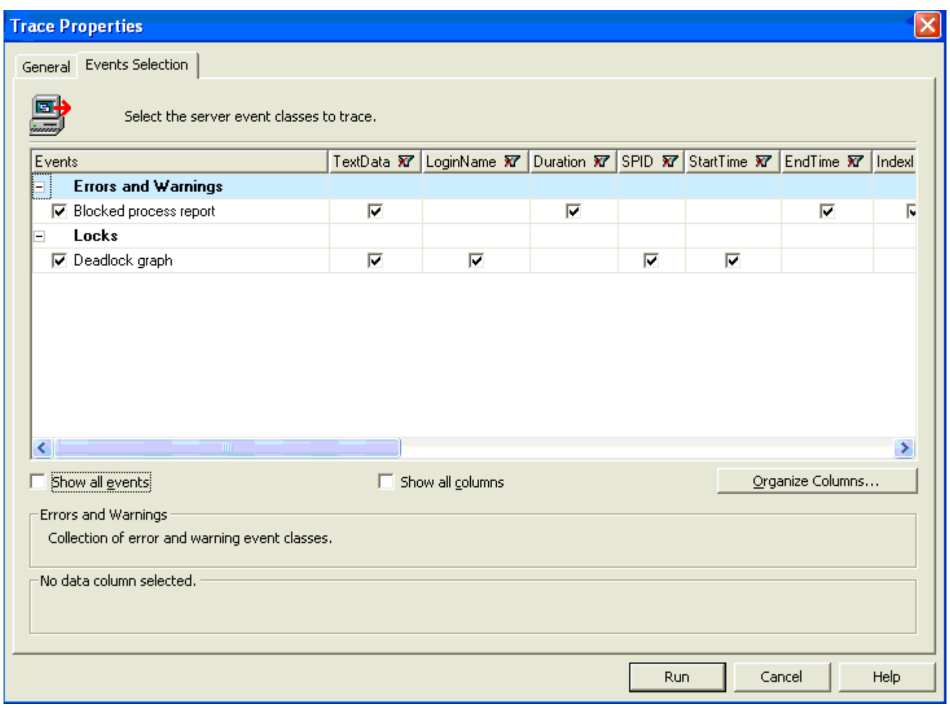


Figure 2: Tracing long blocks and deadlocks

Note This is a light weight trace, as events are only captured when (1) a block exceeds the threshold, or (2) a deadlock occurs. For each 200-second interval that a lock is blocked, a trace event fires. This means that a single lock that is blocked for 600 seconds results in 3 trace events. See Figure 3.

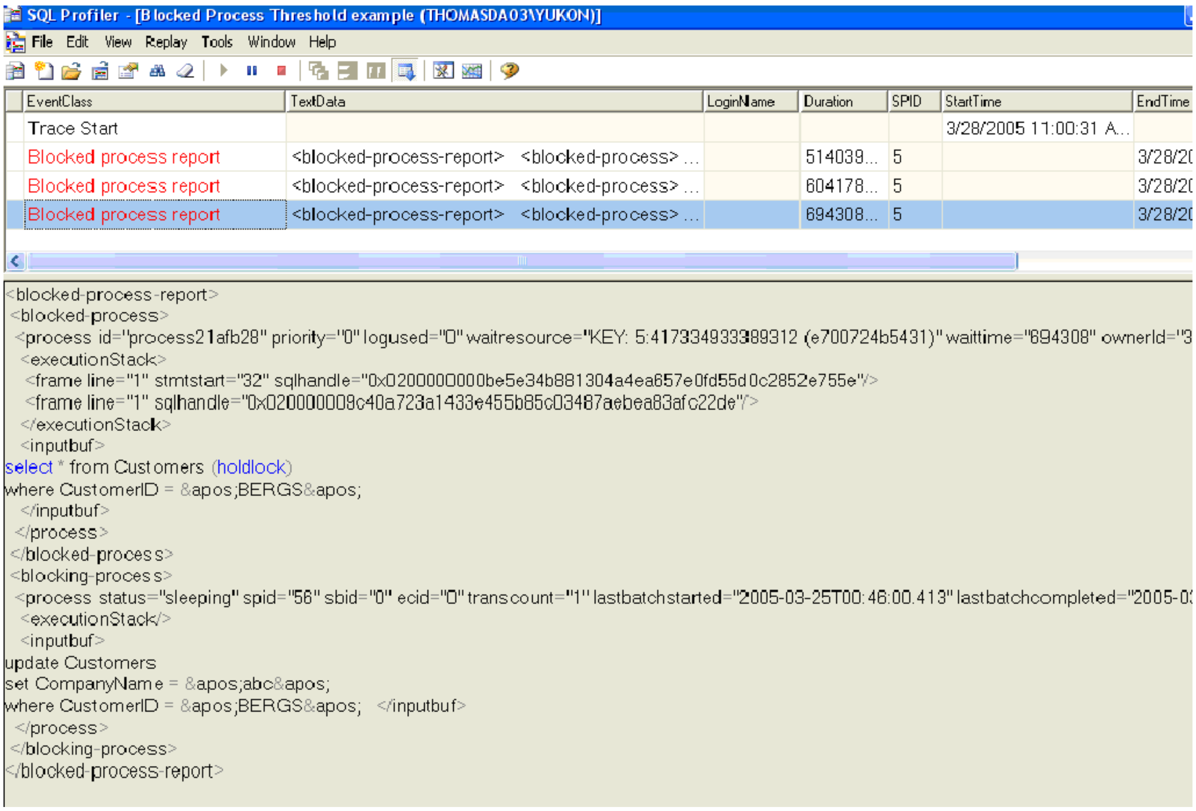


Figure 3: Reporting Blocking > block threshold

The traced event includes the entire SQL statements of both the blocker and the one blocked. In this case the "Update Customers" statement is blocking the "Select from Customers" statement.

By comparison, checking for long blocking scenarios in SQL Server 2000 requires custom code to poll **Sysprocesses** and post-processing the results. Knowledge Base article 271509 contains a sample script that can be used to monitor blocking.

Blocking per object with sys.dm_db_index_operational_stats

The new SQL Server 2005 DMV **Sys.dm_db_index_operational_stats** provides comprehensive index usage statistics, including blocks. In terms of blocking, it provides a detailed accounting of locking statistics per table, index, and partition. Examples of this includes a history of accesses, locks (row_lock_count), blocks (row_lock_wait_count), and waits (row_lock_wait_in_ms) for a given index or table.

The type of information available through this DMV includes:

- Count of locks held, for example, row or page.
- Count of blocks or waits, for example, row, page.
- Duration of blocks or waits, for example, row, page.
- Count of page latches waits.
- Duration of page_latch_wait: This involves contention for a particular page for say, ascending key inserts. In such cases, the hot spot is the last page so multiple writers to the same last page each try to get an exclusive page latch at same time. This will show up as Pagelatch waits.
- Duration of page_io_latch_wait: I/O latch occurs when a user requests a page that is not in the buffer pool. A slow I/O subsystem, or overworked I/O subsystem will sometimes experience high PageIOlatch waits that are actually I/O issues. These issues can be compounded by cache flushes or missing indexes.
- Duration of page latch waits.

Besides blocking related information, there is additional information kept for access to index.

- Types of accesses, for example, range, singleton lookups.
- Inserts, updates, deletes at the leaf level.
- Inserts, updates, deletes above the leaf level. Activity above the leaf is index maintenance. The first row on each leaf page has an entry in the level above. If a new page is allocated at the leaf, the level above will have a new entry for the first row on the new leaf page.
- Pages merged at the leaf level represent empty pages that are de-allocated because rows were deleted.
- Index maintenance. Pages merged above the leaf level are empty pages de-allocated, due to rows deleted at leaf, thereby leaving intermediate level pages empty. The first row on each leaf page has an entry in the level above. If enough rows are deleted at the leaf level, intermediate level index pages that originally contained entries for the first row on leaf pages will be empty. This causes merges to occur above the leaf.

This information is cumulative from instance startup. The information is not retained across instance restarts, and there is no way to reset it. The data returned by this DMV exists only as long as the metadata cache object representing the heap or index is available. The values for each column are set to zero whenever the metadata for the heap or index is brought into the metadata cache. Statistics are accumulated until the cache object is removed from the metadata cache. However, you can periodically poll this table and collect this information in table that can be queried further.

Appendix B defines one such set of stored procedures that can be used to collect index operational data. You can then analyze the data for the time period of interest. Here are the steps to use the stored procedures defined in Appendix B.

1. Initialize the indexstats table by using init_index_operational_stats.
2. Capture a baseline with **insert_indexstats**.
3. Run the workload.
4. Capture the final snapshot of index statistics by using **insert_indexstats**.
5. To analyze the collected index statistics, run the stored procedure **get_indexstats** to generate the average number of locks (Row_lock_count for an index and partition), blocks, and waits per index. A high blocking % and/or high average waits can indicate poor index or query formulation.

Here are some examples that show the kind of information you can get using the above set of stored procedures.

- Get the top five indexes for all databases, order by index usage desc.

```
exec get_indexstats
     @dbid=-1,
     @top='top 5',
     @columns='index, usage',
     @order='index usage'
```

- Get the top five (all columns) index lock promotions where a lock promotion was attempted.

```
exec get_indexstats
       @dbid=-1,
       @top='top 5',
       @order='index lock promotions',
       @threshold='[index lock promotion attempts] > 0'
```

- Get the top five singleton lookups with avg row lock waits>2ms, return columns containing wait, scan, singleton.

```
exec get_indexstats
 @dbid=5,
 @top='top 5',
 @columns='wait,scan,singleton',
 @order='singleton lookups',
 @threshold='[avg row lock wait ms] > 2'
```

- Get the top ten for all databases, columns containing 'avg, wait', order by wait ms, where row lock waits > 1.

```
exec get_indexstats
          @dbid=-1,
          @top='top 10 ',
          @columns='wait,row',
          @order='row lock wait ms',
             @threshold='[row lock waits] > 1'
```

- Get the top five index stats, order by avg row lock waits desc.

```
exec get_indexstats
 @dbid=-1,
 @top='top 5',
 @order='avg row lock wait ms'
```

- Get the top five index stats, order by avg page latch lock waits desc.

```
exec get_indexstats
  @dbid=-1,
  @top='top 5',
  @order='avg page latch wait ms'
```

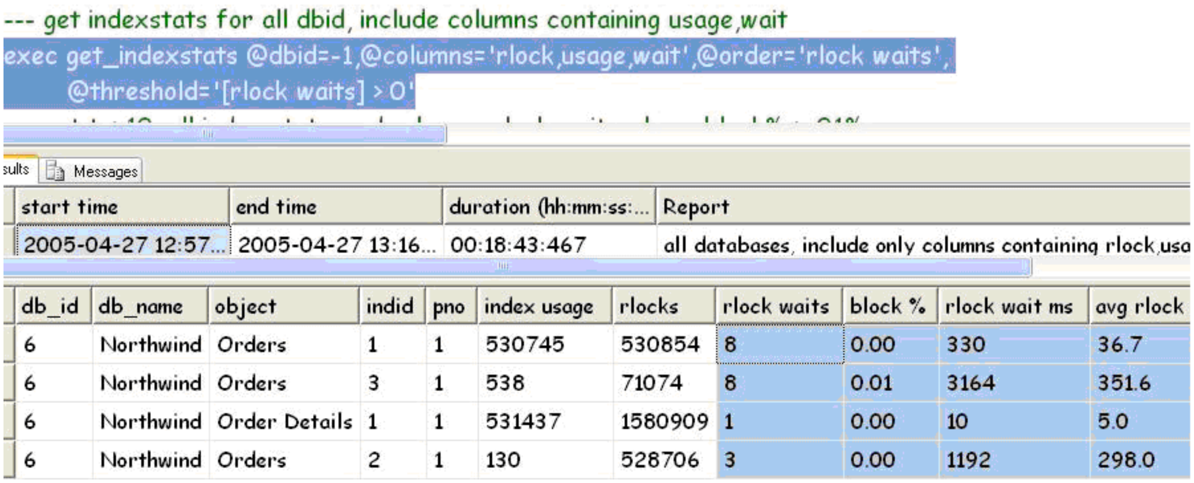
- Get the top 5 percent index stats, order by avg pageio latch waits.

```
exec get_indexstats
  @dbid=-1,
  @top='top 3 percent',
  @order='avg pageio latch wait ms',
  @threshold='[pageio latch waits] > 0'
```

- Get all index stats for the top ten in db=5, ordered by block% where block% > .1.

```
exec get_indexstats
 @dbid=-1,
 @top='top 10',
 @order='block %',
 @threshold='[block %] > 0.1'
```

See the sample Blocking Analysis Report in Figure 4.



```
--- get indexstats for all dbid, include columns containing usage,wait
exec get_indexstats @dbid=-1,@columns='rlock,usage,wait',@order='rlock waits',
      @threshold='[rlock waits] > 0'
```

start time	end time	duration (hh:mm:ss:...	Report
2005-04-27 12:57...	2005-04-27 13:16...	00:18:43:467	all databases, include only columns containing rlock,usa

db_id	db_name	object	indid	pno	index usage	rlocks	rlock waits	block %	rlock wait ms	avg rlock
6	Northwind	Orders	1	1	530745	530854	8	0.00	330	36.7
6	Northwind	Orders	3	1	538	71074	8	0.01	3164	351.6
6	Northwind	Order Details	1	1	531437	1580909	1	0.00	10	5.0
6	Northwind	Orders	2	1	130	528706	3	0.00	1192	298.0

Figure 4: Blocking Analysis Report

SQL Server 2000 does not provide any statistics for the utilization of objects or indexes.

Overall performance effect of blocking using SQL waits

SQL Server 2000 provides 76 wait types for reporting waits. SQL Server 2005 provides over 100 additional wait types for tracking application performance. Any time a user connection is waiting, SQL Server accumulates wait time. For example, the application requests resources such as I/O, locks, or memory and can wait for the resource to be available. This wait information is summarized and categorized across all connections so that a performance profile can be obtained for a given workload. Thus, SQL wait types identify and categorize user (or thread) waits from an application workload or user perspective.

This query lists the top 10 waits in SQL Server. These waits are cumulative but you can reset them using DBCC SQLPERF ([`sys.dm_os_wait_stats`], clear).

```
select top 10 *
from sys.dm_os_wait_stats
order by wait_time_ms desc
```

Following is the output. A few key points to notice are:

- Some waits are normal such as the waits encountered by background threads such as lazy writer.
- Some sessions waited a long time to get a SH lock.
- The *signal wait* is the time between when a worker has been granted access to the resource and the time it gets scheduled on the CPU. A long signal wait may imply high CPU contention.

```
wait_type      waiting_tasks_count   wait_time_ms      max_wait_time_ms
```

```
 signal_wait_time_ms
------------------ -------------------- -------------------- -------------
------- -------
LAZYWRITER_SLEEP      415088              415048437           1812
           156
SQLTRACE_BUFFER_FLUSH 103762              415044000           4000
          0
LCK_M_S               6                   25016812            23240921
          0
WRITELOG              7413                86843               187
          406
LOGMGR_RESERVE_APPEND 82                  82000               1000
          0
SLEEP_BPOOL_FLUSH     4948                28687               31
          15
LCK_M_X               1                   20000               20000
         0
PAGEIOLATCH_SH        871                 11718               140
         15
PAGEIOLATCH_UP        755                 9484                187
         0
IO_COMPLETION         636                 7031                203
         0
```

To analyze the wait states, you need to poll the data periodically and then analyze it later. Appendix B provides the following two sample stored procedures.

- **Track_waitstats**. Collects the data for the desired number of samples and interval between the samples. Here is a sample invocation.

```
exec dbo.track_waitstats @num_samples=6
    ,@delay_interval=30
    ,@delay_type='s'
    ,@truncate_history='y'
                  ,@clear_waitstats='y'
```

- **Get_waitstats**. Analyzes the data collected in the previous steps. Here is a sample invocation.

  ```
  exec [dbo].[get_waitstats_2005]
  ```

  - Spid is running. It then needs an resource that is currently unavailable. Since the resource is not available, it moves to the resource wait list at time T0.
  - A signal indicates that the resource is available, so spid moves to runnable queue at time T1.
  - Spid awaits running status until T2 as cpu works its way through runnable queue in order of arrival.

You can use these stored procedures to analyze the resource waits and signal waits and use this information to isolate the resource contention.

Figure 5 shows a sample report.

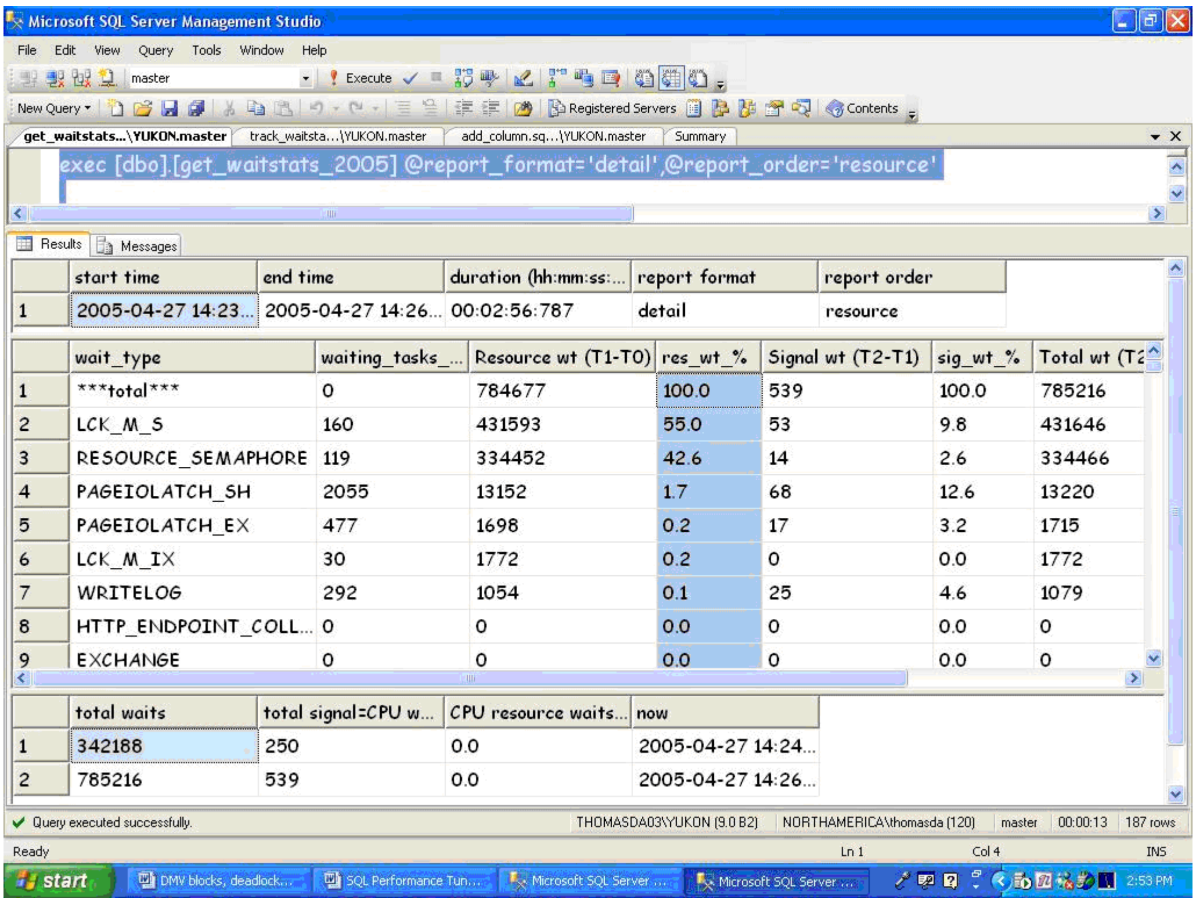


Figure 5: Wait Statistics Analysis Report

The sample Waitstats Analysis Report in Figure 5 indicates a performance problem due to blocking (LCK_M_S) and memory allocation (RESOURCE_SEMAPHORE). Specifically 55% of all waits are for shared locks while 43% are due to memory requests. An analysis of blocking per object will identify the principal points of contention.

Monitoring index usage

Another aspect of query performance is related to DML queries, queries deleting, inserting and modifying data. The more indexes that are defined on a specific table, the more resources are needed to modify data. In combination with locks held over transactions, longer modification operations can hurt concurrency. Therefore, it can be very important to know which indexes are used by an application over time. You can then figure out whether there is a lot of weight in the database schema in the form of indices which never get used.

SQL Server 2005 provides the new **sys.dm_db_index_usage_stats** dynamic management view that shows which indexes are used, and whether they are in use by the user query or only by a system operation. With every execution of a query, the columns in this view are incremented according to the query plan that is used for the execution of that query. The data is collected while SQL Server is up and running. The data in this DMV is kept in memory only and is not persisted. So when the SQL Server instance is shut down, the data is lost. You can poll this table periodically and save the data for later analysis.

The operation on indexes is categorized into user type and system type. User type refers to SELECT and INSERT/DELETE/UPDATE operations. System type operations are commands like DBCC statements or DDL commands or update statistics. The columns for each category of statements differentiate into:

- Seek Operations against an index (user_seeks or system_seeks)
- Lookup Operations against an index (user_lookups or system_lookups)
- Scan Operations against an index (user_scans or system_scans)
- Update Operations against an index (user_updates or system_updates)

For each of these accesses of indexes, the timestamp of the last access is noted as well.

An index itself is identified by three columns covering its database_id, object_id and index_id. Whereas index_id=0 represents a heap table, index_id=1 represents a clustered index whereas index_id>1 represents nonclustered indexes

Over days of runtime of an application against a database, the list of indexes getting accessed in **sys.dm_db_index_usage_stats** will grow.

The rules and definitions for seek, scan, and lookup work as follows in SQL Server 2005.

- SEEK: Indicates the count the B-tree structure was used to access the data. It doesn't matter whether the B-tree structure is used just to read a few pages of each level of the index in order to retrieve one data row or whether half of the index pages are read in order to read gigabytes of data or millions of rows out of the underlying table. So it should be expected that most of the hits against an index are accumulated in this category.
- SCAN: Indicates the count the data layer of the table gets used for retrieval without using one of the index B-trees. In the case of tables that do not have any index defined, this would be the case. In the case of table with indexes defined on it, this can happen when the indexes defined on the table are of no use for the query executed against that statement.

- LOOKUP: Indicates that a clustered index that is defined on a table did get used to look up data which was identified by 'seeking' through a nonclustered index that is defined on that table as well. This describes the scenario known as *bookmark lookup* in SQL Server 2000. It represents a scenario where a nonclustered index is used to access a table and the nonclustered index does not cover the columns of the query select list AND the columns defined in the where clause, SQL Server would increment the value of the column `user_seeks` for the nonclustered index used plus the column `user_lookups` for the entry of the clustered index. This count can become very high if there are multiple nonclustered indexes defined on the table. If the number of `user_seeks` against a clustered index of a table is pretty high, the number of `user_lookups` is pretty high as well plus the number of `user_seeks` of one particular nonclustered index is very high as well, one might be better off by making the nonclustered index with the high count to be the clustered index.

The following DMV query can be used to get useful information about the index usage for all objects in all databases.

```
select object_id, index_id, user_seeks, user_scans, user_lookups

from sys.dm_db_index_usage_stats

order by object_id, index_id
```

One can see the following results for a given table.

```
object_id       index_id    user_seeks    user_scans    user_lookups

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

521690298         1                  0                 251

     123

521690298         2                123                 0

       0
```

In this case, there were 251 executions of a query directly accessing the data layer of the table without using one of the indexes. There were 123 executions of a query accessing the table by using the first nonclustered index, which does not cover either the select list of the query or the columns specified in the WHERE clause since we see 123 lookups on the clustered index.

The most interesting category to look at is the 'user type statement' category. Usage indication in the 'system category' can be seen as a result of the existence of the index. If the index did not exist, it would not have to be updated in statistics and it would not need to be checked for consistency. Therefore, the analysis needs to focus on the four columns that indicate usage by ad hoc statements or by the user application.

To get information about the indexes of a specific table that has not been used since the last start of SQL Server, this query can be executed in the context of the database that owns the object.

```
select i.name
from sys.indexes i
where i.object_id=object_id('<table_name>') and
    i.index_id NOT IN  (select s.index_id
                        from sys.dm_db_index_usage_stats s
                        where s.object_id=i.object_id and
                        i.index_id=s.index_id and
                        database_id = <dbid> )
```

All indexes which haven't been used yet can be retrieved with the following statement:

```
select object_name(object_id), i.name
from sys.indexes i
where  i.index_id NOT IN (select s.index_id
                          from sys.dm_db_index_usage_stats s
                          where s.object_id=i.object_id and
                          i.index_id=s.index_id and
                          database_id = <dbid> )
order by object_name(object_id) asc
```

In this case, the table name and the index name are sorted according to the table name.

The real purpose of this dynamic management view is to observe the usage of indexes in the long run. It might make sense to take a snapshot of that view or a snapshot of the result of the query and store it away every day to compare the changes over time. If you can identify that particular indexes did not get used for months or during periods such as quarter-end reporting or fiscal-year reporting, you could eventually delete those indexes from the database.

Conclusion

For more information:

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

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), how would you rate this paper?

Appendix A: DBCC MEMORYSTATUS Description

There is some information that is primarily available by using the DBCC MEMORYSTATUS command. However, some of this information is also available using the dynamic management views (DMVs).

SQL Server 2000 DBCC MEMORYSTATUS is described at

http://support.microsoft.com/?id=271624

SQL Server 2005 DBCC MEMORYSTATUS command is described at

http://support.microsoft.com/?id=907877

Appendix B: Blocking Scripts

This appendix provides the source listing of the stored procedures referred to in this white paper. You can use these stored procedures as they are or tailor them to suit your needs.

sp_block

```
create proc dbo.sp_block (@spid bigint=NULL)
as
-- This stored procedure is provided "AS IS" with no warranties, and
-- confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- T. Davidson
-- This proc reports blocks
--    1. optional parameter @spid
--



select
    t1.resource_type,
    'database'=db_name(resource_database_id),
    'blk object' = t1.resource_associated_entity_id,
    t1.request_mode,
    t1.request_session_id,
    t2.blocking_session_id
from
    sys.dm_tran_locks as t1,
    sys.dm_os_waiting_tasks as t2
where
    t1.lock_owner_address = t2.resource_address and
    t1.request_session_id = isnull(@spid,t1.request_session_id)
```

Analyzing operational index statistics

This set of stored procedures can be used to analyze index usage.

get_indexstats

```
create proc dbo.get_indexstats
    (@dbid smallint=-1
    ,@top varchar(100)=NULL
    ,@columns varchar(500)=NULL
    ,@order varchar(100)='lock waits'
    ,@threshold varchar(500)=NULL)
as
--
-- This stored procedure is provided "AS IS" with no warranties, and
-- confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- T. Davidson
-- This proc analyzes index statistics including accesses, overhead,
-- locks, blocks, and waits
--
-- Instructions: Order of execution is as follows:
--    (1) truncate indexstats with init_indexstats
--    (2) take initial index snapshot using insert_indexstats
--    (3) Run workload
--    (4) take final index snapshot using insert_indexstats
--    (5) analyze with get_indexstats

-- @dbid limits analysis to a database
-- @top allows you to specify TOP n
-- @columns is used to specify what columns from
--               sys.dm_db_index_operational_stats will be included in
-- the report
--               For example, @columns='scans,lookups,waits' will include
-- columns
--               containing these keywords
```

```
-- @order used to order results
-- @threshold used to add a threshold,
--                 example: @threshold='[block %] > 5' only include if
-- blocking is over 5%
--
------  definition of some computed columns returned
-- [blk %] = percentage of locks that cause blocks e.g. blk% = 100 * lock
-- waits / locks
-- [index usage] = range_scan_count + singleton_lookup_count +
-- leaf_insert_count
-- [nonleaf index overhead]=nonleaf_insert_count + nonleaf_delete_count +
-- nonleaf_update_count
-- [avg row lock wait ms]=row_lock_wait_in_ms/row_lock_wait_count
-- [avg page lock wait ms]=page_lock_wait_in_ms/page_lock_wait_count
-- [avg page latch wait ms]=page_latch_wait_in_ms/page_latch_wait_count
-- [avg pageio latch wait
-- ms]=page_io_latch_wait_in_ms/page_io_latch_wait_count
--------------------------------------------------------------------------
-------------------------
--- Case 1 - only one snapshot of sys.dm_db_operational_index_stats was
-- stored in
---               indexstats. This is an error - return errormsg to user
--- Case 2 - beginning snapshot taken, however some objects were not
-- referenced
---               at the time of the beginning snapshot. Thus, they will
-- not be in the initial
---               snapshot of sys.dm_db_operational_index_stats, use 0 for
-- starting values.
---               Print INFO msg for informational purposes.
--- Case 3 - beginning and ending snapshots, beginning values for all
-- objects and indexes
---               this should be the normal case, especially if SQL Server
-- is up a long time
--------------------------------------------------------------------------
-------------------------
set nocount on
```

```
declare @orderby varchar(100), @where_dbid_is varchar(100), @temp
varchar(500), @threshold_temptab varchar(500)
declare @cmd varchar(max),@col_stmt varchar(500),@addcol varchar(500)
declare @begintime datetime, @endtime datetime, @duration datetime,
@mincount int, @maxcount int

select @begintime = min(now), @endtime = max(now) from indexstats

if @begintime = @endtime
    begin
        print 'Error: indexstats contains only 1 snapshot of
sys.dm_db_index_operational_stats'
        print 'Order of execution is as follows: '
        print '    (1) truncate indexstats with init_indexstats'
        print '    (2) take initial index snapshot using insert_indexstats'
        print '    (3) Run workload'
        print '    (4) take final index snapshot using insert_indexstats'
        print '    (5) analyze with get_indexstats'
        return -99
    end

select @mincount = count(*) from indexstats where now = @begintime
select @maxcount = count(*) from indexstats where now = @endtime

if @mincount < @maxcount
    begin
        print 'InfoMsg1: sys.dm_db_index_operational_stats only contains
entries for objects referenced since last SQL re-cycle'
        print 'InfoMsg2: Any newly referenced objects and indexes captured
in the ending snapshot will use 0 as a beginning value'
    end

select @top = case
        when @top is NULL then ''
        else lower(@top)
    end,
```

```
        @where_dbid_is = case (@dbid)
        when -1 then ''
        else ' and i1.database_id = ' + cast(@dbid as varchar(10))
    end,
--- thresholding requires a temp table
        @threshold_temptab = case
        when @threshold is NULL then ''
        else ' select * from #t where ' + @threshold
    end
--- thresholding requires temp table, add 'into #t' to select statement
select @temp = case (@threshold_temptab)
        when '' then ''
        else ' into #t '
    end
select @orderby=case(@order)
when 'leaf inserts' then 'order by [' + @order + ']'
when 'leaf deletes' then 'order by [' + @order + ']'
when 'leaf updates' then 'order by [' + @order + ']'
when 'nonleaf inserts' then 'order by [' + @order + ']'
when 'nonleaf deletes' then 'order by [' + @order + ']'
when 'nonleaf updates' then 'order by [' + @order + ']'
when 'nonleaf index overhead' then 'order by [' + @order + ']'
when 'leaf allocations' then 'order by [' + @order + ']'
when 'nonleaf allocations' then 'order by [' + @order + ']'
when 'allocations' then 'order by [' + @order + ']'
when 'leaf page merges' then 'order by [' + @order + ']'
when 'nonleaf page merges' then 'order by [' + @order + ']'
when 'range scans' then 'order by [' + @order + ']'
when 'singleton lookups' then 'order by [' + @order + ']'
when 'index usage' then 'order by [' + @order + ']'
when 'row locks' then 'order by [' + @order + ']'
when 'row lock waits' then 'order by [' + @order + ']'
when 'block %' then 'order by [' + @order + ']'
when 'row lock wait ms' then 'order by [' + @order + ']'
when 'avg row lock wait ms' then 'order by [' + @order + ']'
when 'page locks' then 'order by [' + @order + ']'
```

```
when 'page lock waits' then 'order by [' + @order + ']'
when 'page lock wait ms' then 'order by [' + @order + ']'
when 'avg page lock wait ms' then 'order by [' + @order + ']'
when 'index lock promotion attempts' then 'order by [' + @order + ']'
when 'index lock promotions' then 'order by [' + @order + ']'
when 'page latch waits' then 'order by [' + @order + ']'
when 'page latch wait ms' then 'order by [' + @order + ']'
when 'pageio latch waits' then 'order by [' + @order + ']'
when 'pageio latch wait ms' then 'order by [' + @order + ']'
else ''
end

if @orderby <> '' select @orderby = @orderby + ' desc'
select
    'start time'=@begintime,
    'end time'=@endtime,
    'duration (hh:mm:ss:ms)'=convert(varchar(50),
    @endtime-@begintime,14),
    'Report'=case (@dbid)
               when -1 then 'all databases'
               else db_name(@dbid)
             end +

            case
                when @top = '' then ''
                when @top is NULL then ''
                when @top = 'none' then ''
                else ', ' + @top
            end +
            case
                when @columns = '' then ''
                when @columns is NULL then ''
                when @columns = 'none' then ''
                else ', include only columns containing ' + @columns
            end +
            case(@orderby)
```

```
                when '' then ''
                when NULL then ''
                when 'none' then ''
                else ', ' + @orderby
            end +
            case
                when @threshold = '' then ''
                when @threshold is NULL then ''
                when @threshold = 'none' then ''
                else ', threshold on ' + @threshold
            end

select @cmd = ' select i2.database_id, i2.object_id, i2.index_id,
i2.partition_number '
select @cmd = @cmd +' , begintime=case min(i1.now) when max(i2.now) then
NULL else min(i1.now) end '
select @cmd = @cmd +'   , endtime=max(i2.now) '
select @cmd = @cmd +' into #i '
select @cmd = @cmd +' from indexstats i2 '
select @cmd = @cmd +' full outer join '
select @cmd = @cmd +'   indexstats i1 '
select @cmd = @cmd +' on i1.database_id = i2.database_id '
select @cmd = @cmd +' and i1.object_id = i2.object_id '
select @cmd = @cmd +' and i1.index_id = i2.index_id '
select @cmd = @cmd +' and i1.partition_number = i2.partition_number '
select @cmd = @cmd +' where i1.now >= ''' +
convert(varchar(100),@begintime, 109) + ''''
select @cmd = @cmd +' and i2.now = ''' + convert(varchar(100),@endtime,
109) + ''''
select @cmd = @cmd + ' ' + @where_dbid_is + ' '
select @cmd = @cmd + ' group by i2.database_id, i2.object_id, i2.index_id,
i2.partition_number '
select @cmd = @cmd + ' select ' + @top + ' i.database_id,
db_name=db_name(i.database_id),
object=isnull(object_name(i.object_id),i.object_id), indid=i.index_id,
part_no=i.partition_number '
```

```
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[leaf inserts]=i2.leaf_insert_count -
         isnull(i1.leaf_insert_count,0)'

select @cmd = @cmd +@addcol
exec dbo.add_column
     @add_stmt=@addcol out,
     @cols_containing=@columns,@col_stmt=' ,
       [leaf deletes]=i2.leaf_delete_count –
       isnull(i1.leaf_delete_count,0)'

select @cmd = @cmd + @addcol

exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[leaf updates]=i2.leaf_update_count –
isnull(i1.leaf_update_count,0)'

select @cmd = @cmd + @addcol

exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[nonleaf inserts]=i2.nonleaf_insert_count –
isnull(i1.nonleaf_insert_count,0)'

select @cmd = @cmd + @addcol

exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[nonleaf deletes]=i2.nonleaf_delete_count –
```

```
isnull(i1.nonleaf_delete_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[nonleaf updates]=i2.nonleaf_update_count - 
isnull(i1.nonleaf_update_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[nonleaf index overhead]=(i2.nonleaf_insert_count - 
isnull(i1.nonleaf_insert_count,0)) + (i2.nonleaf_delete_count - 
isnull(i1.nonleaf_delete_count,0)) + (i2.nonleaf_update_count - 
isnull(i1.nonleaf_update_count,0))'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[leaf allocations]=i2.leaf_allocation_count - 
isnull(i1.leaf_allocation_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[nonleaf allocations]=i2.nonleaf_allocation_count - 
isnull(i1.nonleaf_allocation_count,0)'

select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
```

```
    @col_stmt=' ,[allocations]=(i2.leaf_allocation_count -
isnull(i1.leaf_allocation_count,0)) + (i2.nonleaf_allocation_count -
isnull(i1.nonleaf_allocation_count,0))'

select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[leaf page merges]=i2.leaf_page_merge_count -
isnull(i1.leaf_page_merge_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[nonleaf page merges]=i2.nonleaf_page_merge_count -
isnull(i1.nonleaf_page_merge_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[range scans]=i2.range_scan_count -
isnull(i1.range_scan_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing= @columns,
    @col_stmt=' ,[singleton lookups]=i2.singleton_lookup_count -
isnull(i1.singleton_lookup_count,0)'

select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
```

```
    @col_stmt=' ,[index usage]=(i2.range_scan_count -
isnull(i1.range_scan_count,0)) + (i2.singleton_lookup_count -
isnull(i1.singleton_lookup_count,0)) + (i2.leaf_insert_count -
isnull(i1.leaf_insert_count,0))'
select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[row locks]=i2.row_lock_count -
isnull(i1.row_lock_count,0)'
select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[row lock waits]=i2.row_lock_wait_count -
isnull(i1.row_lock_wait_count,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[block %]=cast (100.0 * (i2.row_lock_wait_count -
isnull(i1.row_lock_wait_count,0)) / (1 + i2.row_lock_count -
isnull(i1.row_lock_count,0)) as numeric(5,2))'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[row lock wait ms]=i2.row_lock_wait_in_ms -
isnull(i1.row_lock_wait_in_ms,0)'

select @cmd = @cmd + @addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
```

```
    @col_stmt=' ,[avg row lock wait ms]=cast ((1.0*(i2.row_lock_wait_in_ms
- isnull(i1.row_lock_wait_in_ms,0)))/(1 + i2.row_lock_wait_count -
isnull(i1.row_lock_wait_count,0)) as numeric(20,1))'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[page locks]=i2.page_lock_count -
isnull(i1.page_lock_count,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[page lock waits]=i2.page_lock_wait_count -
isnull(i1.page_lock_wait_count,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[page lock wait ms]=i2.page_lock_wait_in_ms -
isnull(i1.page_lock_wait_in_ms,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[avg page lock wait ms]=cast
((1.0*(i2.page_lock_wait_in_ms - isnull(i1.page_lock_wait_in_ms,0)))/(1 +
i2.page_lock_wait_count - isnull(i1.page_lock_wait_count,0)) as
numeric(20,1))'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[index lock promotion
attempts]=i2.index_lock_promotion_attempt_count -
isnull(i1.index_lock_promotion_attempt_count,0)'
```

```
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[index lock promotions]=i2.index_lock_promotion_count -
isnull(i1.index_lock_promotion_count,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[page latch waits]=i2.page_latch_wait_count -
isnull(i1.page_latch_wait_count,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[page latch wait ms]=i2.page_latch_wait_in_ms -
isnull(i1.page_latch_wait_in_ms,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[avg page latch wait ms]=cast
((1.0*(i2.page_latch_wait_in_ms - isnull(i1.page_latch_wait_in_ms,0)))/(1
+ i2.page_latch_wait_count - isnull(i1.page_latch_wait_count,0)) as
numeric(20,1))'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[pageio latch waits]=i2.page_io_latch_wait_count -
isnull(i1.page_latch_wait_count,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
```

```
    @col_stmt=' ,[pageio latch wait ms]=i2.page_io_latch_wait_in_ms -
isnull(i1.page_latch_wait_in_ms,0)'
select @cmd = @cmd +@addcol
exec dbo.add_column
    @add_stmt=@addcol out,
    @cols_containing=@columns,
    @col_stmt=' ,[avg pageio latch wait ms]=cast
((1.0*(i2.page_io_latch_wait_in_ms -
isnull(i1.page_io_latch_wait_in_ms,0)))/(1 + i2.page_io_latch_wait_count -
isnull(i1.page_io_latch_wait_count,0)) as numeric(20,1))'

select @cmd = @cmd +@addcol
select @cmd = @cmd + @temp
select @cmd = @cmd + ' from #i i '
select @cmd = @cmd + ' left join indexstats i1 on i.begintime = i1.now and
i.database_id = i1.database_id and i.object_id = i1.object_id and
i.index_id = i1.index_id and i.partition_number = i1.partition_number '

select @cmd = @cmd + ' left join indexstats i2 on i.endtime = i2.now and
i.database_id = i2.database_id and i.object_id = i2.object_id and
i.index_id = i2.index_id and i.partition_number = i2.partition_number '
select @cmd = @cmd + ' ' + @orderby + ' '
select @cmd = @cmd + @threshold_temptab
exec ( @cmd )
go
```

insert_indexstats

```
create proc insert_indexstats (@dbid smallint=NULL,
                               @objid int=NULL,
                               @indid int=NULL,
                               @partitionid int=NULL)
as
--
-- This stored procedure is provided "AS IS" with no warranties, and
confers no rights.
-- Use of included script samples are subject to the terms specified at
http://www.microsoft.com/info/cpyright.htm
-- This stored procedure stores a snapshot of
sys.dm_db_index_operational_stats into the table indexstas
-- for later analysis by the stored procedure get_indexstats. Please note
that the indexstats table has an additional
-- column to store the timestamp when the snapshot is taken
--
-- T. Davidson
-- snapshot sys.dm_db_index_operational_stats
--
declare @now datetime
select @now = getdate()
insert into indexstats
        (database_id
        ,object_id
        ,index_id
        ,partition_number
        ,leaf_insert_count
        ,leaf_delete_count
        ,leaf_update_count
        ,leaf_ghost_count
        ,nonleaf_insert_count
        ,nonleaf_delete_count
        ,nonleaf_update_count
        ,leaf_allocation_count
```

```
        ,nonleaf_allocation_count
        ,leaf_page_merge_count
        ,nonleaf_page_merge_count
        ,range_scan_count
        ,singleton_lookup_count
        ,forwarded_fetch_count
        ,lob_fetch_in_pages
        ,lob_fetch_in_bytes
        ,lob_orphan_create_count
        ,lob_orphan_insert_count
        ,row_overflow_fetch_in_pages
        ,row_overflow_fetch_in_bytes
        ,column_value_push_off_row_count
        ,column_value_pull_in_row_count
        ,row_lock_count
        ,row_lock_wait_count
        ,row_lock_wait_in_ms
        ,page_lock_count
        ,page_lock_wait_count
        ,page_lock_wait_in_ms
        ,index_lock_promotion_attempt_count
        ,index_lock_promotion_count
        ,page_latch_wait_count
        ,page_latch_wait_in_ms
        ,page_io_latch_wait_count
        ,page_io_latch_wait_in_ms,
        now)
select  database_id
        ,object_id
        ,index_id
        ,partition_number
        ,leaf_insert_count
        ,leaf_delete_count
        ,leaf_update_count
        ,leaf_ghost_count
        ,nonleaf_insert_count
```

```
        ,nonleaf_delete_count
        ,nonleaf_update_count
        ,leaf_allocation_count
        ,nonleaf_allocation_count
        ,leaf_page_merge_count
        ,nonleaf_page_merge_count
        ,range_scan_count
        ,singleton_lookup_count
        ,forwarded_fetch_count
        ,lob_fetch_in_pages
        ,lob_fetch_in_bytes
        ,lob_orphan_create_count
        ,lob_orphan_insert_count
        ,row_overflow_fetch_in_pages
        ,row_overflow_fetch_in_bytes
        ,column_value_push_off_row_count
        ,column_value_pull_in_row_count
        ,row_lock_count
        ,row_lock_wait_count
        ,row_lock_wait_in_ms
        ,page_lock_count
        ,page_lock_wait_count
        ,page_lock_wait_in_ms
        ,index_lock_promotion_attempt_count
        ,index_lock_promotion_count
        ,page_latch_wait_count
        ,page_latch_wait_in_ms
        ,page_io_latch_wait_count
        ,page_io_latch_wait_in_ms
        ,@now
from sys.dm_db_index_operational_stats(@dbid,@objid,@indid,@partitionid)
go
```

init_index_operational_stats

```
CREATE proc dbo.init_index_operational_stats
as
--
-- This stored procedure is provided "AS IS" with no warranties, and
-- confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- T. Davidson
--
-- create indexstats table if it doesn't exist, otherwise truncate
--
set nocount on
if not exists (select 1 from dbo.sysobjects where
id=object_id(N'[dbo].[indexstats]') and OBJECTPROPERTY(id, N'IsUserTable')
= 1)
    create table dbo.indexstats (
        database_id smallint NOT NULL
        ,object_id int NOT NULL
        ,index_id int NOT NULL
        ,partition_number int NOT NULL
        ,leaf_insert_count bigint NOT NULL
        ,leaf_delete_count bigint NOT NULL
        ,leaf_update_count bigint NOT NULL
        ,leaf_ghost_count bigint NOT NULL
        ,nonleaf_insert_count bigint NOT NULL
        ,nonleaf_delete_count bigint NOT NULL
        ,nonleaf_update_count bigint NOT NULL
        ,leaf_allocation_count bigint NOT NULL
        ,nonleaf_allocation_count bigint NOT NULL
        ,leaf_page_merge_count bigint NOT NULL
        ,nonleaf_page_merge_count bigint NOT NULL
        ,range_scan_count bigint NOT NULL
        ,singleton_lookup_count bigint NOT NULL
```

```
        ,forwarded_fetch_count bigint NOT NULL
        ,lob_fetch_in_pages bigint NOT NULL
        ,lob_fetch_in_bytes bigint NOT NULL
        ,lob_orphan_create_count bigint NOT NULL
        ,lob_orphan_insert_count bigint NOT NULL
        ,row_overflow_fetch_in_pages bigint NOT NULL
        ,row_overflow_fetch_in_bytes bigint NOT NULL
        ,column_value_push_off_row_count bigint NOT NULL
        ,column_value_pull_in_row_count bigint NOT NULL
        ,row_lock_count bigint NOT NULL
        ,row_lock_wait_count bigint NOT NULL
        ,row_lock_wait_in_ms bigint NOT NULL
        ,page_lock_count bigint NOT NULL
        ,page_lock_wait_count bigint NOT NULL
        ,page_lock_wait_in_ms bigint NOT NULL
        ,index_lock_promotion_attempt_count bigint NOT NULL
        ,index_lock_promotion_count bigint NOT NULL
        ,page_latch_wait_count bigint NOT NULL
        ,page_latch_wait_in_ms bigint NOT NULL
        ,page_io_latch_wait_count bigint NOT NULL
        ,page_io_latch_wait_in_ms bigint NOT NULL
        ,now datetime default getdate())
else  truncate table dbo.indexstats
go
```

add_column

```
create proc dbo.add_column (
            @add_stmt varchar(500) output,
            @find varchar(100)=NULL,
            @cols_containing varchar(500)=NULL,
            @col_stmt varchar(max))
as
--
-- This stored procedure is provided "AS IS" with no warranties, and
-- confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- T. Davidson
-- @add_stmt is the result passed back to the caller
-- @find is a keyword from @cols_containing
-- @cols_containing is the list of keywords to include in the report
-- @col_stmt is the statement that will be compared with @find.
--                If @col_stmt contains @find, include this statement.
--                set @add_stmt = @col_stmt
--
declare @length int, @strindex int, @EOS bit
if @cols_containing is NULL
    begin
        select @add_stmt=@col_stmt
        return
    end
select @add_stmt = '', @EOS = 0

while @add_stmt is not null and @EOS = 0
            @dbid=-1,
    select @strindex = charindex(',',@cols_containing)
    if @strindex = 0
            select @find = @cols_containing, @EOS = 1
    else
```

```
    begin
        select @find = substring(@cols_containing,1,@strindex-1)
        select @cols_containing =
            substring(@cols_containing,
                      @strindex+1,
                      datalength(@cols_containing) - @strindex)
    end
    select @add_stmt=case
--when @cols_containing is NULL then NULL
    when charindex(@find,@col_stmt) > 0 then NULL
    else ''
    end
end
--- NULL indicates this statement is to be passed back through out parm
@add_stmt
if @add_stmt is NULL select @add_stmt=@col_stmt
go
```

Wait states

This set of stored procedures can be used to analyze the blocking in SQL Server.

track_waitstats_2005

```
CREATE proc [dbo].[track_waitstats_2005] (
                            @num_samples int=10,
                            @delay_interval int=1,
                            @delay_type nvarchar(10)='minutes',
                            @truncate_history nvarchar(1)='N',
                            @clear_waitstats nvarchar(1)='Y')
as
--
-- This stored procedure is provided "AS IS" with no warranties, and
-- confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- T. Davidson
-- @num_samples is the number of times to capture waitstats, default is 10
-- times
-- default delay interval is 1 minute
-- delaynum is the delay interval - can be minutes or seconds
-- delaytype specifies whether the delay interval is minutes or seconds
-- create waitstats table if it doesn't exist, otherwise truncate
-- Revision: 4/19/05
--- (1) added object owner qualifier
--- (2) optional parameters to truncate history and clear waitstats
set nocount on
if not exists (select 1
               from sys.objects
               where object_id = object_id ( N'[dbo].[waitstats]') and
               OBJECTPROPERTY(object_id, N'IsUserTable') = 1)
    create table [dbo].[waitstats]
        ([wait_type] nvarchar(60) not null,
        [waiting_tasks_count] bigint not null,
        [wait_time_ms] bigint not null,
```

```
        [max_wait_time_ms] bigint not null,
        [signal_wait_time_ms] bigint not null,
        now datetime not null default getdate())

If lower(@truncate_history) not in (N'y',N'n')
    begin
        raiserror ('valid @truncate_history values are ''y'' or
''n''',16,1) with nowait
    end
If lower(@clear_waitstats) not in (N'y',N'n')
    begin
        raiserror ('valid @clear_waitstats values are ''y'' or
''n''',16,1) with nowait
    end
If lower(@truncate_history) = N'y'
    truncate table dbo.waitstats

If lower (@clear_waitstats) = N'y'
    -- clear out waitstats
    dbcc sqlperf ([sys.dm_os_wait_stats],clear) with no_infomsgs

declare @i int,
        @delay varchar(8),
        @dt varchar(3),
        @now datetime,
        @totalwait numeric(20,1),
        @endtime datetime,
        @begintime datetime,
        @hr int,
        @min int,
        @sec int

select @i = 1
select @dt = case lower(@delay_type)
    when N'minutes' then 'm'
    when N'minute' then 'm'
```

```
        when N'min' then 'm'
        when N'mi' then 'm'
        when N'n' then 'm'
        when N'm' then 'm'
        when N'seconds' then 's'
        when N'second' then 's'
        when N'sec' then 's'
        when N'ss' then 's'
        when N's' then 's'
        else @delay_type
    end

    if @dt not in ('s','m')
    begin
        raiserror ('delay type must be either ''seconds'' or
    ''minutes''',16,1) with nowait
        return
    end
    if @dt = 's'
    begin
        select @sec = @delay_interval % 60, @min = cast((@delay_interval / 60)
    as int), @hr = cast((@min / 60) as int)
    end
    if @dt = 'm'
    begin
        select @sec = 0, @min = @delay_interval % 60, @hr =
    cast((@delay_interval / 60) as int)
    end
    select @delay= right('0'+ convert(varchar(2),@hr),2) + ':' +
        + right('0'+convert(varchar(2),@min),2) + ':' +
        + right('0'+convert(varchar(2),@sec),2)

    if @hr > 23 or @min > 59 or @sec > 59
    begin
        select 'delay interval and type: ' + convert
    (varchar(10),@delay_interval) + ',' + @delay_type + ' converts to ' +
```

```
@delay
    raiserror ('hh:mm:ss delay time cannot > 23:59:59',16,1) with nowait
    return
end
while (@i <= @num_samples)
begin
    select @now = getdate()
    insert into [dbo].[waitstats] (
                        [wait_type],
                        [waiting_tasks_count],
                        [wait_time_ms],
                        [max_wait_time_ms],
                        [signal_wait_time_ms],
                        now)
            select
                [wait_type],
                [waiting_tasks_count],
                [wait_time_ms],
                [max_wait_time_ms],
                [signal_wait_time_ms],
                @now
            from sys.dm_os_wait_stats

    insert into [dbo].[waitstats] (
                        [wait_type],
                        [waiting_tasks_count],
                        [wait_time_ms],
                        [max_wait_time_ms],
                        [signal_wait_time_ms],
                        now)
            select
                'Total',
                sum([waiting_tasks_count]),
                sum([wait_time_ms]),
                0,
                sum([signal_wait_time_ms]),
```

```
                @now
            from [dbo].[waitstats]
            where now = @now

        select @i = @i + 1
        waitfor delay @delay
    end
    --- create waitstats report
    execute dbo.get_waitstats_2005
    go
    exec dbo.track_waitstats @num_samples=6
        ,@delay_interval=30
        ,@delay_type='s'
        ,@truncate_history='y'
        ,@clear_waitstats='y'
```

get_waitstats_2005

```
CREATE proc [dbo].[get_waitstats_2005] (
                @report_format varchar(20)='all',
                @report_order varchar(20)='resource')
as
-- This stored procedure is provided "AS IS" with no warranties, and
-- confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- this proc will create waitstats report listing wait types by
-- percentage.
--    (1) total wait time is the sum of resource & signal waits,
--                @report_format='all' reports resource & signal
--    (2) Basics of execution model (simplified)
--        a. spid is running then needs unavailable resource, moves to
--           resource wait list at time T0
--        b. a signal indicates resource available, spid moves to
--           runnable queue at time T1
--        c. spid awaits running status until T2 as cpu works its way
```

```
--           through runnable queue in order of arrival
--      (3) resource wait time is the actual time waiting for the
--          resource to be available, T1-T0
--      (4) signal wait time is the time it takes from the point the
--          resource is available (T1)
--          to the point in which the process is running again at T2.
--          Thus, signal waits are T2-T1
--      (5) Key questions: Are Resource and Signal time significant?
--          a. Highest waits indicate the bottleneck you need to solve
--             for scalability
--          b. Generally if you have LOW% SIGNAL WAITS, the CPU is
--             handling the workload e.g. spids spend move through
--             runnable queue quickly
--          c. HIGH % SIGNAL WAITS indicates CPU can't keep up,
--             significant time for spids to move up the runnable queue
--             to reach running status
--      (6) This proc can be run when track_waitstats is executing
--
-- Revision 4/19/2005
-- (1) add computation for CPU Resource Waits = Sum(signal waits /
--                                                    total waits)
-- (2) add @report_order parm to allow sorting by resource, signal
--     or total waits
--
set nocount on

declare @now datetime,
        @totalwait numeric(20,1),
        @totalsignalwait numeric(20,1),
        @totalresourcewait numeric(20,1),
        @endtime datetime,@begintime datetime,
        @hr int,
        @min int,
        @sec int

if not exists (select 1
```

```
                from sysobjects
                where id = object_id ( N'[dbo].[waitstats]') and
                      OBJECTPROPERTY(id, N'IsUserTable') = 1)
begin
        raiserror('Error [dbo].[waitstats] table does not exist',
                 16, 1) with nowait
        return
end

if lower(@report_format) not in ('all','detail','simple')
    begin
        raiserror ('@report_format must be either ''all'',
                    ''detail'', or ''simple''',16,1) with nowait
        return
    end
if lower(@report_order) not in ('resource','signal','total')
    begin
        raiserror ('@report_order must be either ''resource'',
            ''signal'', or ''total''',16,1) with nowait
        return
    end
if lower(@report_format) = 'simple' and lower(@report_order) <> 'total'
    begin
        raiserror ('@report_format is simple so order defaults to
''total''',
                        16,1) with nowait
        select @report_order = 'total'
    end


select
    @now=max(now),
    @begintime=min(now),
    @endtime=max(now)
from [dbo].[waitstats]
where [wait_type] = 'Total'
```

```
--- subtract waitfor, sleep, and resource_queue from Total
select @totalwait = sum([wait_time_ms]) + 1, @totalsignalwait =
sum([signal_wait_time_ms]) + 1
from waitstats
where [wait_type] not in (
        'CLR_SEMAPHORE',
        'LAZYWRITER_SLEEP',
        'RESOURCE_QUEUE',
        'SLEEP_TASK',
        'SLEEP_SYSTEMTASK',
        'Total' ,'WAITFOR',
        '***total***') and
    now = @now

select @totalresourcewait = 1 + @totalwait - @totalsignalwait

-- insert adjusted totals, rank by percentage descending
delete waitstats
where [wait_type] = '***total***' and
now = @now

insert into waitstats
select
    '***total***',
    0,@totalwait,
    0,
    @totalsignalwait,
    @now

select 'start time'=@begintime,'end time'=@endtime,
       'duration (hh:mm:ss:ms)'=convert(varchar(50),@endtime-
@begintime,14),
       'report format'=@report_format, 'report order'=@report_order

if lower(@report_format) in ('all','detail')
```

```
begin
----- format=detail, column order is resource, signal, total. order by
resource desc
    if lower(@report_order) = 'resource'
        select [wait_type],[waiting_tasks_count],
            'Resource wt (T1-T0)'=[wait_time_ms]-[signal_wait_time_ms],
            'res_wt_%'=cast (100*([wait_time_ms] -
                    [signal_wait_time_ms]) /@totalresourcewait as
numeric(20,1)),
            'Signal wt (T2-T1)'=[signal_wait_time_ms],
            'sig_wt_%'=cast (100*[signal_wait_time_ms]/@totalsignalwait as
numeric(20,1)),
            'Total wt (T2-T0)'=[wait_time_ms],
            'wt_%'=cast (100*[wait_time_ms]/@totalwait as numeric(20,1))
        from waitstats
        where [wait_type] not in (
                'CLR_SEMAPHORE',
                'LAZYWRITER_SLEEP',
                'RESOURCE_QUEUE',
                'SLEEP_TASK',
                'SLEEP_SYSTEMTASK',
                'Total',
                'WAITFOR') and
                now = @now
        order by 'res_wt_%' desc

----- format=detail, column order signal, resource, total. order by signal
desc
    if lower(@report_order) = 'signal'
        select    [wait_type],
                [waiting_tasks_count],
                'Signal wt (T2-T1)'=[signal_wait_time_ms],
                'sig_wt_%'=cast
(100*[signal_wait_time_ms]/@totalsignalwait as numeric(20,1)),
                'Resource wt (T1-T0)'=[wait_time_ms]-
[signal_wait_time_ms],
```

```
                'res_wt_%'=cast (100*([wait_time_ms] -
                        [signal_wait_time_ms]) /@totalresourcewait as
numeric(20,1)),
                'Total wt (T2-T0)'=[wait_time_ms],
                'wt_%'=cast (100*[wait_time_ms]/@totalwait as
numeric(20,1))
        from waitstats
        where [wait_type] not in (
                    'CLR_SEMAPHORE',
                    'LAZYWRITER_SLEEP',
                    'RESOURCE_QUEUE',
                    'SLEEP_TASK',
                    'SLEEP_SYSTEMTASK',
                    'Total',
                    'WAITFOR') and
                    now = @now
        order by 'sig_wt_%' desc

----- format=detail, column order total, resource, signal. order by total
 desc
    if lower(@report_order) = 'total'
        select
            [wait_type],
            [waiting_tasks_count],
            'Total wt (T2-T0)'=[wait_time_ms],
            'wt_%'=cast (100*[wait_time_ms]/@totalwait as numeric(20,1)),
            'Resource wt (T1-T0)'=[wait_time_ms]-[signal_wait_time_ms],
            'res_wt_%'=cast (100*([wait_time_ms] -
                    [signal_wait_time_ms]) /@totalresourcewait as
numeric(20,1)),
            'Signal wt (T2-T1)'=[signal_wait_time_ms],
            'sig_wt_%'=cast (100*[signal_wait_time_ms]/@totalsignalwait as
 numeric(20,1))
        from waitstats
        where [wait_type] not in (
                'CLR_SEMAPHORE',
```

```
                'LAZYWRITER_SLEEP',
                'RESOURCE_QUEUE',
                'SLEEP_TASK',
                'SLEEP_SYSTEMTASK',
                'Total',
                'WAITFOR') and
                now = @now
        order by 'wt_%' desc
end
else
---- simple format, total waits only
    select
        [wait_type],
        [wait_time_ms],
        percentage=cast (100*[wait_time_ms]/@totalwait as numeric(20,1))
    from waitstats
    where [wait_type] not in (
                    'CLR_SEMAPHORE',
                    'LAZYWRITER_SLEEP',
                    'RESOURCE_QUEUE',
                    'SLEEP_TASK',
                    'SLEEP_SYSTEMTASK',
                    'Total',
                    'WAITFOR') and
                now = @now
    order by percentage desc


---- compute cpu resource waits
select
    'total waits'=[wait_time_ms],
    'total signal=CPU waits'=[signal_wait_time_ms],
    'CPU resource waits % = signal waits / total waits'=
            cast (100*[signal_wait_time_ms]/[wait_time_ms] as
 numeric(20,1)),
    now
```

```
from [dbo].[waitstats]
where [wait_type] = '***total***'
order by now
go


declare @now datetime
select @now = getdate()
select getdate()
```