Troubleshooting Performance Problems in SQL Server 2008

SQL Server Technical Article

Writers: Sunil Agarwal, Boris Baryshnikov, Keith Elmore, Juergen Thomas, Kun Cheng,

Burzin Patel

Technical Reviewers: Jerome Halmans, Fabricio Voznika, George Reynya

Published: March 2009

Applies to: SQL Server 2008

Summary: Sometimes a poorly designed database or a system that is improperly configured for the workload can cause the slowdowns in SQL Server. Administrators need to proactively prevent or minimize problems and, when they occur, diagnose the cause and take corrective action. This paper provides step-by-step guidelines for diagnosing and troubleshooting common performance problems by using publicly available tools such as SQL Server Profiler, Performance Monitor, dynamic management views, and SQL Server Extended Events (Extended Events) and the data collector, which are new in SQL Server 2008.

Copyright

Table of Contents

Introduction

It’s not uncommon to experience the occasional slowdown of a database running the Microsoft® SQL Server® database software. 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; if they occur, you want to diagnose the cause and take corrective actions to fix the problem whenever possible. This white 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 (in the Windows Server® 2003 operating system) or Performance Monitor (in the Windows Vista® operating system and Windows Server 2008), also known as Perfmon; dynamic management views (sometimes referred to as DMVs); and SQL Server Extended Events (Extended Events) and the data collector, which are new in SQL Server 2008. We have limited the scope of this white paper to the problems commonly seen by Microsoft Customer Service and Support, because an exhaustive analysis of all possible problems is not feasible.

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 2008 has made great strides in supportability. New dynamic management views (DMVs) have been added, like **sys.dm_os_memory_brokers**, **sys.dm_os_memory_nodes**, and **sys.dm_exec_procedure_stats**. Existing DMVs such as **sys._dm_os_sys_info**, **sys.dm_exec_requests**, and **sys.dm_exec_requests** have been enriched with additional information. You can use DMVs and existing tools, like SQL Server Profiler and Performance Monitor, to collect performance related data for analysis.

The secondary goal of this paper is to introduce new troubleshooting tools and features in SQL Server 2008, including Extended Events and the data collector.

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**: Because there is only one **tempdb** for each SQL Server instance, it can be a performance and a disk space bottleneck. An application can overload **tempdb** through excessive DDL or DML operations and by taking too much 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 might regress, or a new query might 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 can be due to poor application or schema design or the choice of 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 if the query optimizer decides not to use it, the query can slow down; these conditions also put 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-run query can put pressure on the CPU.

New Performance Tools in SQL Server 2008

SQL Server 2008 introduced new features and tools that you can use to monitor and troubleshoot performance problems. We'll discuss two features: Extended Events and the data collector.

Resource Bottlenecks

The next sections of this paper discuss 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 whether 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 (at a time 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 or memory, or you may need to increase the bandwidth of your I/O or network channel. However, before you take that step, it is useful to understand some common causes of resource bottlenecks. Some solutions, such as reconfiguration, do not require the addition of more resources.

Tools for Resolving Resource Bottlenecks

One or more of the following tools can be used to resolve a particular resource bottleneck:

- **Performance Monitor**: This tool is available as part of the Windows® operating system. For more information, see your Windows documentation.
- **SQL Server Profiler**: See **SQL Server Profiler** in the **Performance Tools** group in the **SQL Server 2008** program group. For more information, see SQL Server 2008 Books Online.
- **DBCC commands**: For more information, see SQL Server 2008 Books Online and Appendix A.

- **DMVs**: For more information, see SQL Server 2008 Books Online.
- **Extended Events**: For more information, see Extended Events later in this paper and SQL Server 2008 Books Online.
- **Data collector and the management data warehouse (MDW)**: For more information, see Data Collector and the MDW later in this paper and SQL Server 2008 Books Online.

CPU Bottlenecks

A CPU bottleneck can be caused by hardware resources that are insufficient for the load. However, excessive CPU utilization can commonly be reduced by query tuning (especially if there was a sudden increase without additional load or different queries on the server), addressing any application design factors, and optimizing the system configuration. Before you rush out to buy faster and/or more processors, identify the largest consumers of CPU bandwidth and see whether you can tune those queries or adjust the design/configuration factors.

Performance Monitor is generally one of the easiest means to determine whether the server is CPU bound. You should look to see whether the **Processor:% Processor Time** counter is high; sustained values in excess of 80% of the processor time per CPU are generally deemed to be a bottleneck.

From within SQL Server, you can also check for CPU bottlenecks by checking the DMVs. Requests waiting with the SOS_SCHEDULER_YIELD wait type or a high number of runnable tasks can indicate that runnable threads are waiting to be scheduled and that there might be a CPU bottleneck on the processor. If you have enabled the data collector, the SQL Server Waits chart on the Server Activity report is a very easy way to monitor for CPU bottlenecks over time. Consumed CPU and SOS_SCHEDULER_YIELD waits are rolled up into the CPU Wait Category in this report, and if you do see high CPU utilization, you can drill through to find the queries that are consuming the most resources.

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 for detecting and resolving these problems.

Excessive Query Compilation and Optimization

Query compilation and optimization is a CPU-intensive process. The cost of optimization increases as the complexity of the query and the underlying schema increases, but even a relatively simply query can take 10-20 milliseconds of CPU time to parse and compile.

To mitigate this cost, SQL Server caches and reuses compiled query plans. Each time a new query is received from the client, SQL Server first searches the plan cache (sometimes referred to as the procedure cache) to see whether there is already a compiled plan that can be reused. If a matching query plan cannot be found, SQL Server parses and compiles the incoming query before running it.

For an OLTP-type workload, the set of queries that are submitted is relatively small and static. Quite commonly the optimal query plan does not depend on the exact value or values used as predicates in the query because the lookups are based on keys. Reusing query plans in this type of workload is very important because the cost of compilation may be as high as or higher than the cost of executing the query itself. However, a data-warehousing workload may benefit greatly from using ad hoc SQL and letting the query optimizer search for the optimal plan for each set of values, because the run time for these queries is typically much longer than the compile time, and the optimal query plan is more likely to change depending on the predicates in the query. Using parameterized queries or stored procedures for OLTP-based applications substantially increases the chance of reusing a cached plan and can result in substantial reductions in SQL Server CPU consumption. You can enable parameterization at the database or query level by using the PARAMETERIZATION FORCED database option or query hint, respectively. For more information about important limitations, especially if you rely on indexes on computed columns or indexed views, see SQL Server 2008 Books Online.

However, the best place to parameterize queries is within the application itself (at design time), which also helps mitigate the risk of SQL injection by avoiding string concatenation using parameter values. For more information, see the following topics in SQL Server 2008 Books Online:

- SQL Injection (http://msdn.microsoft.com/en-us/library/ms161953.aspx)
- Using sp_executesql (http://msdn.microsoft.com/en-us/library/ms175170.aspx)

Detection

During compilation, SQL Server 2008 computes a “signature” of the query and exposes this as the **query_hash** column in **sys.dm_exec_requests** and **sys.dm_exec_query_stats**, and the QueryHash attribute in Showplan/Statistics XML. Entities with the same **query_hash** value have a high probability of referring to the same query text if it had been written in a **query_hash** parameterized form. Queries that vary only in literal values should have the same value. For example, the first two queries share the same query hash, while the third query has a different query hash, because it is performing a different operation.

```
select * from sys.objects where object_id = 100
select * from sys.objects where object_id = 101
select * from sys.objects where name = 'sysobjects'
```

The query hash is computed from the tree structure produced during compilation. Whitespace is ignored, as are differences in the use of explicit column lists compared to using an asterisk (*) in the SELECT list. Furthermore, it does not matter if one query uses fully qualified name and another uses just the table name as long as they both refer to the same object. All of the following should produce the same **query_hash** value.

```
Use AdventureWorks
Go

set showplan_xml on
go

-- Assume this is run by a user whose default schema is Sales
select * from SalesOrderHeader h

select * from Sales.SalesOrderHeader h

select SalesOrderID,
      RevisionNumber,
      OrderDate,
      DueDate,
      ShipDate,
      Status,
      OnlineOrderFlag,
      SalesOrderNumber,
      PurchaseOrderNumber,
      AccountNumber,
      CustomerID,
      ContactID,
      SalesPersonID,
      TerritoryID,
      BillToAddressID,
      ShipToAddressID,
      ShipMethodID,
      CreditCardID,
      CreditCardApprovalCode,
      CurrencyRateID,
      SubTotal,
      TaxAmt,
      Freight,
      TotalDue,
      Comment,
      rowguid,
      ModifiedDate
from Sales.SalesOrderHeader h
go
set showplan_xml off
go
```

Note that the database portion of the fully qualified name is ignored when the query_hash value is generated. This allows resource usage to be aggregated across all queries in systems that replicate the same schema and queries against many databases on the same instance.

An easy way to detect applications that submit lots of ad hoc queries is by grouping on the **sys.dm_exec_query_stats.query_hash** column as follows.

```
select q.query_hash,
      q.number_of_entries,
      t.text as sample_query,
      p.query_plan as sample_plan
from (select top 20 query_hash,
                  count(*) as number_of_entries,
                  min(sql_handle) as sample_sql_handle,
                  min(plan_handle) as sample_plan_handle
            from sys.dm_exec_query_stats
            group by query_hash
            having count(*) > 1
            order by count(*) desc) as q
      cross apply sys.dm_exec_sql_text(q.sample_sql_handle) as t
      cross apply sys.dm_exec_query_plan(q.sample_plan_handle) as p
go
```

Queries that have a number_of_entries value in the hundreds or thousands are excellent candidates for parameterization. If you look at the CompileTime and CompileCPU attributes under the <QueryPlan> tag of the sample XML query plan and multiply those values times the number_of_entries value for that query, you can get an estimate of how much compile time and CPU you can eliminate by parameterizing the query (which means that the query is compiled once, and then it is cached and reused for subsequent executions). Eliminating these unnecessary cached plans has other intangible benefits as well, such as freeing memory to cache other compiled plans (thereby further reducing compilation overhead) and leaving more memory for the buffer cache.

Resolution

SQL Server 2008 also produces a **query_plan_hash** value that represents a "signature" of the query plan's access path (that is, what join algorithm is used, the join order, index selection, and so forth). Some applications might rely on getting a different query plan based on the optimizer evaluating the specific parameter values passed to that execution of the query. If that is the case, you do not want to parameterize the queries.

You can use the **query_hash** and **query_plan_hash** values together to determine whether a set of ad hoc queries with the same query_hash value resulted in query plans with the same or different **query_plan_hash** values, or access path. This is done via a small modification to the earlier query.

```
select q.query_hash,
      q.number_of_entries,
      q.distinct_plans,
      t.text as sample_query,
      p.query_plan as sample_plan
from (select top 20 query_hash,
                  count(*) as number_of_entries,
                  count(distinct query_plan_hash) as distinct_plans,
                  min(sql_handle) as sample_sql_handle,
                  min(plan_handle) as sample_plan_handle
            from sys.dm_exec_query_stats
            group by query_hash
            having count(*) > 1
            order by count(*) desc) as q
      cross apply sys.dm_exec_sql_text(q.sample_sql_handle) as t
      cross apply sys.dm_exec_query_plan(q.sample_plan_handle) as p
go
```

Note that this new query returns a count of the number of distinct query plans (**query_plan_hash** values) for a given **query_hash** value. Rows that return a large number for **number_of_entries** and a **distinct_plans** count of 1 are good candidates for parameterization. Even if the number of distinct plans is more than one, you can use **sys.dm_exec_query_plan** to retrieve the different query plans and examine them to see whether the difference is important and necessary for achieving optimal performance.

After you determine which queries should be parameterized, the best place to parameterize them is the client application. The details of how you do this vary slightly depending on which client API you use, but the one consistent thing across all of the APIs is that instead of building the query string with literal predicates, you build a string with a question mark (?) as a parameter marker.

```
      -- Submitting as ad hoc query
      select * from Sales.SalesOrderHeader where SalesOrderID = 100

      -- Submitting as parameterized
      select * from Sales.SalesOrderHeader where SalesOrderID = ?
```

You should use the appropriate APIs for your technology (ODBC, OLE DB, or SQLClient) to bind a value to the parameter marker. The client driver or provider then submits the query in its parameterized form using **sp_executesql**.

```
exec sp_executesql N'select * from Sales.SalesOrderHeader where
SalesOrderID = @P1', N'@P1 int', 100
```

Because the query is parameterized, it matches and reuses an existing cached plan.

If the entire workload for a given database is appropriate for parameterization and you do not have control over (or can't change) the client application, you can also enable the forced parameterization option for the database. Note the caveats mentioned earlier; this can prevent the optimizer from matching indexed views and indexes on computed columns.

```
ALTER DATABASE AdventureWorks SET PARAMETERIZATION FORCED
```

If you can't parameterize the client application or enable forced parameterization for the entire database, you can still create a template plan guide for specific queries with the OPTION (PARAMETERIZATION FORCED) hint. For more information about the steps required to do this, see Forced Parameterization (http://technet.microsoft.com/en-us/library/ms175037.aspx) in SQL Server 2008 Books Online.

Unnecessary Recompilation

When a batch or remote procedure call (RPC) is submitted to SQL Server, the server checks for the validity and correctness of the query plan before it begins executing. 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 recompilation. In SQL Server 2008 and SQL Server 2005, the behavior is changed to statement-level recompilation of stored procedures. When SQL Server 2008 or 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 in response to various conditions, such as:

- Schema changes
- Statistics changes
- Deferred compilation
- SET option changes

- Temporary table changes
- Stored procedure creation with the RECOMPILE query hint or the OPTION (RECOMPILE) query hint

Detection

You can use Performance Monitor and SQL Server Profiler to detect excessive compilation and recompilation.

Performance Monitor

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 whether the compilations 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.

These are the key data counters:

- 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 (http://msdn.microsoft.com/en-us/library/ms190911.aspx) in SQL Server 2008 Books Online.

SQL Server Profiler Trace

If the Performance Monitor counters indicate a high number of recompilations, the recompilations could be contributing to the high CPU consumed by SQL Server. Look at the profiler trace to find the stored procedures that are being recompiled. The SQL Server Profiler trace provides 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. Some of the more important data columns for the **SP:Recompile** event class are listed here. The **EventSubClass** data column in particular is important for determining the reason for the recompilation. **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 2008 and SQL Server 2005, it is more useful to monitor **SQL:StmtRecompile**, because this event class is fired when any type of batch, ad hoc, stored procedure, or trigger is recompiled.

The key data columns to 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 (http://technet.microsoft.com/en-us/library/ms179294.aspx) in SQL Server 2008 Books Online.

If you have a trace file saved, you can use the following query to see all the recompilation 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

For more information about trace events, see sp_trace_setevent (http://msdn.microsoft.com/en-us/library/ms186265.aspx) in SQL Server 2008 Books Online.

You could further group the results from this query by the **SqlHandle** and **ObjectID** columns, or by various other columns, to see whether most of the recompilations 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 SQL Server compiles or recompiles a Transact-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, because it is captured for each compilation or recompilation. If you see a high value for the **SQL Compilations/sec** counter in Performance 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 recompilations.

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. Because the elapsed time during optimization is generally close to the CPU time that is used for the optimization (because the optimization process is very CPU bound), you can get a good measure of the extent to which the compilation and recompilation 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 to look at are as follows:

- **Sql_handle**
- **Total worker time**
- **Plan generation number**
- **Statement Start Offset**

For more information, see sys.dm_exec_query_stats (http://msdn.microsoft.com/en-us/library/ms189741.aspx) in SQL Server 2008 Books Online.

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

Resolution

If you detect excessive compilation and recompilation, consider the following options:

- If the recompilation 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 recompilations 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 less optimal 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 displays "Statistics Changed" for an operation on a temporary table.
- To avoid recompilations that are due to changes in statistics (for example, if 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 to ensure correctness (for example, when the underlying table structure has changed and the plan no longer applies) and not to respond to changes in statistics. For example, a recompilation can occur 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 recompilations that are due to statistics changes on that object. Note, however, that turning off the auto-stats feature by using this method is not usually 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 recompilations that are due to deferred compilations, 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 whether any indexing changes improve the compile time and the execution time of the query.
- Check to see whether the stored procedure was created with the WITH RECOMPILE option or whether the RECOMPILE query hint was used. If a procedure was created with the WITH RECOMPILE option, in SQL Server 2008 or SQL Server 2005, you may be able to take advantage of a statement-level RECOMPILE hint if a particular statement within that procedure needs to be recompiled. Using this hint at the statement level avoids 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 about the RECOMPILE hint, see Query Hints (Transact-SQL)

(http://msdn.microsoft.com/en-us/library/ms181714.aspx) in SQL Server 2008 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. If read-ahead (prefetch) is 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. If these types of operations are no longer constrained by physical I/O, they tend to manifest themselves in 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). 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 can cause increased CPU consumption.

The following 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, you can 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 detect inefficient query plans:

- Tune the query with the Database Engine Tuning Advisor to see whether 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 whether 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, you can use the plan guide feature to specify query hints for queries that match certain text criteria. Plan guides can be created for ad hoc queries as well as queries inside a stored procedure. Hints such as OPTION (OPTIMIZE FOR) enable 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) provide you with varying degrees of control over the query plan. SQL Server 2008 offers full

DML support for plan guides, which means that that they can be created for SELECT, INSERT, UPDATE, DELETE or MERGE statements.

- SQL Server 2008 also offers a new feature called plan freezing that allows you to freeze a plan exactly as it exists in the plan cache. This option is similar to creating a plan guide with the USE PLAN query hint specified. However, it eliminates the need to execute lengthy commands as required when creating a plan guides. It also minimizes the user errors with go along with those lengthy commands. For example, the simple two-statement batch presented below is all that's needed to freeze a plan for a query that matches the specified text criteria.

```
DECLARE @plan_handle varbinary(64);

-- Extract the query's plan_handle.
SELECT @plan_handle = plan_handle FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(sql_handle)
WHERE text LIKE N'Some query matching criteria%';

EXECUTE sp_create_plan_guide_from_handle
    @name =  N'Sample_PG1',
    @plan_handle = @plan_handle,
    @statement_start_offset = NULL;
GO
```

This statement creates a plan guide (Sample_PG1) in the **sys.plan_guides** table.

Intraquery 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, 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 by using the **max degree of parallelism** option, on a resource workload group level, or on a per-query level by 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 determines how many schedulers are underutilized and chooses a DOP for the query that fully utilizes the remaining schedulers. After 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 amount of 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 might choose to run with a parallel plan and use a DOP of four and spawn up threads on four different processors. After those threads start executing, existing connections can 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 useful in a transaction processing environment.

SQL Server 2008 implemented significant scalability improvements to fully utilize available hardware with partitioned table queries. Consequently, SQL Server 2008 might use higher amounts of CPU during parallel query execution than older versions. If this is not desired, you should limit or disable parallelism.

Detection

Intraquery parallelism problems can be detected by using the following methods.

Performance Monitor

For more information, see the **SQL Server:SQL Statistics – Batch Requests/sec** counter and SQL Statistics Object (http://msdn.microsoft.com/en-us/library/ms190911.aspx) in SQL Server 2008 Books Online.

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, you can easily retrieve the text of the query by using **sys.dm_exec_sql_text**, and you can retrieve the plan by using **sys.dm_exec_cached_plan**.

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

```
--
-- Find query plans that can 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 concurrently use CPU. Note that not all parallel queries demonstrate this behavior (where the CPU time is 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 oOr 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 identifies as expensive enough to exceed the cost threshold of parallelism, which defaults to 5 (roughly a 5-second execution time on a reference computer). Any queries identified through the previous methods are candidates for further tuning.
- Use the Database Engine Tuning Advisor to see whether 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, because 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 multistatement table-valued functions or CLR functions, table variables, or comparisons with a Transact-SQL variable (comparisons with a parameter are okay).
    - 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, because it could have other pending results. A common solution was to change the connection properties to use a server-side cursor.

When a server-side cursor is used, the database client software (the OLE DB provider or ODBC driver) transparently encapsulates client requests inside special extended stored procedures, such as **sp_cursoropen** or **sp_cursorfetch**. This is referred to as an *API cursor* (as opposed to a Transact-SQL 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 result in a **sp_cursorfetch** instructing the server to send back only a certain number of rows. By controlling the number of rows that are fetched, the ODBC driver or OLE DB provider can cache the row or rows. 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 can 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 that return the same 100 rows one row at a time.

Detection

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

Performance Monitor

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. Systems that have high CPU utilization because of small fetch sizes typically have hundreds of cursor requests per second. There are no specific counters that list the fetch buffer size.

DMVs

You can use following query to determine the connections with API cursors (as opposed to Transact-SQL 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 (Transact-SQL cursors
always
have a 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 whether 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 2008.
- 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 the Windows operating system, each process has its own virtual address space (sometimes referred to as VAS). The set of all virtual addresses available for process use constitutes the size of the virtual address space. The size of the virtual address space depends on the architecture (32-bit 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, especially on 32-bit platform while physical memory may still be available.

For more information about virtual address space, see Process Address Space (http://technet.microsoft.com/en-us/library/ms189334.aspx) in SQL Server 2008 Books Online and Virtual Address Space (http://msdn.microsoft.com/library/en-us/memory/base/virtual_address_space.asp) in the Win32® and COM Development documentation in the MSDN® Library.

AWE, Locked Pages, and SQL Server

Address Windowing Extensions (AWE) is an API that allows a 32-bit application to manipulate physical memory beyond 4 GB memory limit. The 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-bit 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 might have detrimental effects. For this reason, in order to use AWE, the **Lock Pages in Memory** privilege must be granted for the account that runs SQL Server.

From a troubleshooting perspective, an important point is that the SQL Server buffer pool uses AWE allocated memory; however, only database (hashed) pages can mapped or unmapped to take advantage of using additional 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 counters that are specific to SQL Server counters or dynamic management views to obtain this information.

For more information about AWE mapped memory, see Managing Memory for Large Databases (http://msdn.microsoft.com/en-us/library/ms191481.aspx) and Memory Architecture (http://msdn.microsoft.com/en-us/library/ms187499.aspx) in SQL Server 2008 Books Online, and Large Memory Support (http://msdn.microsoft.com/en-us/library/aa366718.aspx) in the Win32 and COM Development documentation in the MSDN Library. For more information about Physical Address Extension (PAE), 4-gigabyte tuning (/3GB, /USERVA boot options), and AWE, see Physical Address Extension (http://msdn.microsoft.com/en-us/library/aa366796.aspx) in the Win32 and COM Development documentation in the MSDN Library.

The following table summarizes the maximum memory support options for different configurations of SQL Server 2008. (Note that a particular edition of SQL Server or Windows may put more restrictive limits on the amount of supported memory.)

Table 1

Configuration	User VAS	Max physical memory	AWE/locked pages support
Native 32-bit on 32-bit operating system	2 GB	64 GB	Yes
with /3GB boot parameter[1]	3 GB	16 GB	Yes
32-bit on x64 operating system (WOW)	4 GB	64 GB	Yes
32-bit on IA64 operating system (WOW)	2 GB	2 GB	No
Native 64-bit on x64 operating system	8 terabytes	2 terabytes	Yes
Native 64-bit on IA64 operating system	7 terabytes	2 terabytes	Yes

Current maximum memory limit depends on the Windows edition and service pack. For more information, see Memory Limits for Windows Releases (http://msdn.microsoft.com/en-us/library/aa366778.aspx) in the Win32 and COM Development documentation in the MSDN Library.

[1] For more information about boot parameters, see Using AWE (http://technet.microsoft.com/en-us/library/ms175581.aspx) in SQL Server 2008 Books Online.

Memory Pressures

Memory pressure indicates that a limited amount of memory is available. Identifying when SQL Server runs under 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 time-out 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 can result in overall slowdown. SQL Server detects this condition and, depending on the configuration, it might 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 (for example, **max server memory**). • Changes in memory distribution of internal components.
Virtual	Running low on available memory commitment (the difference between **Memory:Commit Limit** and **Memory:Committed Bytes**) is low (could be due to lack of space in the system page files). This can cause the system to fail memory allocations. This condition can result in the whole system responding very slowly or even bring it to a halt.	Running low on VAS (direct allocations, DLLs loaded in SQL Server VAS, high number of threads) or VAS fragmentation (a lot of VAS is available but in small blocks). SQL Server detects this condition and can release reserved regions of VAS, reduce buffer pool commit target, and start shrinking caches.

Windows has a notification mechanism that reports if physical memory is running high or low. SQL Server uses this mechanism in its memory management decisions. For more information, see QueryMemoryResourceNotification in the Win32 and COM Development documentation in the MSDN Library.

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

Table 3

Pressure	External	Internal
Physical	1. Find major system memory consumers. 2. Attempt to eliminate (if possible). 3. Check for adequate system RAM and consider adding more RAM (usually requires more careful investigation beyond the scope of this paper).	1. Identify major memory consumers inside SQL Server. 2. Verify server configuration. 3. Further actions depend on the investigation: check for workload; possible design issues; other resource bottlenecks.
Virtual	1. Increase swap file size. 2. Check for major physical memory consumers and follow steps of external physical memory pressure.	Follow steps of internal physical memory pressure.

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 for the server. However, signs of memory pressure can also indicate that the server is running close to its capacity and the potential for out-of-memory errors exists. In the case of a normally operating server, you can use information about memory pressures as a baseline for determining reasons for out-of-memory conditions later.

Tools for Memory Diagnostics

The following tools and sources of information can be used for memory troubleshooting:

- General system and SQL Server state and memory-specific DMVs.
- The DBCC MEMORYSTATUS command.
- SQL Server ring buffers.
- Performance counters.
- The SQL Server error log, and Windows application and system logs. You can use Log File Viewer in SQL Server Management Studio to look at Windows and SQL Server error logs in one place in a time-synchronized fashion. Log File Viewer is accessible through Object Explorer. For connected database servers, expand **Management**, and then click **SQL Server Logs**. For more information, see Log File Viewer (http://msdn.microsoft.com/en-us/library/dd206996.aspx)in SQL Server 2008 Books Online.

New DMVs in SQL Server 2008

Several new dynamic management views (sometimes known as DMVs) in SQL Server 2008 simplify retrieval of information that can be helpful with memory troubleshooting.

In some cases, newly introduced DMVs provide information that was previously available only in DBCC MEMORYSTATUS output. The following list provides a summary of new DMVs for memory troubleshooting:

- **sys.dm_os_memory_brokers** provides information about memory allocations using the internal SQL Server memory manager. The information provided can be useful in determining very large memory consumers.
- **sys.dm_os_memory_nodes** and **sys.dm_os_memory_node_access_stats** provide information about physical non-uniform memory access (NUMA) memory nodes and node access statistics grouped by the type of the page. (**sys.dm_os_memory_node_access_stats** is populated under dynamic trace flag 842 due to its performance impact.)
- **sys.dm_os_nodes** provides information about CPU node configuration for SQL Server. This DMV also reflects software NUMA (soft-NUMA) configuration.
- **sys.dm_os_process_memory** provides overview information about SQL Server memory usage.
- **sys.dm_os_sys_memory** provides overview information about the system memory usage.
- **sys.dm_resource_governor_configuration**, **sys.dm_resource_governor_resource_pools**, and **sys.dm_resource_governor_workload_groups** provide information about the state of the Resource Governor feature of SQL Server 2008. Some of the configuration parameters of Resource Governor affect how SQL Server allocates memory; you should check these parameters during memory troubleshooting.

For more information about these DMVs, see SQL Server 2008 Books Online.

Resource Governor in SQL Server 2008

Resource Governor is a new feature of SQL Server 2008 Enterprise that enables you to modify SQL Server memory allocation policies. When you troubleshoot memory-related errors, remember that out-of-memory errors can be related to the configuration of Resource Governor. Specifically:

- The out-of-memory error message (701) now always includes a reference to the Resource Governor resource pool, regardless of whether Resource Governor is used or present or not. This reference does not necessarily indicate a problem with your Resource Governor configuration; it refers to the fact that the failed memory allocation happened as part of particular resource pool (which can be the "internal" predefined resource pool on editions where Resource Governor is not present).
- Reducing MAX_MEMORY_PERCENT for a Resource Governor resource pool can cause out-of-memory errors in the pool even if memory is otherwise available.
- MIN_MEMORY_PERCENT values greater than zero on some resource pools can lower the amount of memory that is available to other resource pools, causing effects that are similar to lowering MAX_MEMORY_PERCENT. For more information about effective maximums, see Resource Governor Concepts (http://msdn.microsoft.com/en-us/library/bb934084.aspx) in SQL Server 2008 Books Online.

- There is a set of memory brokers for each Resource Governor resource pool, which are reflected in the output of the **sys.dm_os_memory_brokers** DMV and the output of DBCC MEMORYSTATUS. Internal memory pressure might be localized to a particular Resource Governor resource pool, depending on the load and configuration of that pool. Memory usage of query compilation, execution, and caches is partitioned on a per-resource-pool basis.
- Commands such as DBCC FREESYSTEMCACHE and DBCC FREEPROCCACHE take **pool_id** as a parameter, enabling part of the cache to be cleared, corresponding to a particular resource pool.
- A set of DMVs that you can use in memory troubleshooting (**sys.dm_exec_cached_plans**, **sys.dm_os_memory_cache_entries**, **sys.dm_exec_query_resource_semaphores**, **sys.dm_exec_query_memory_grants**, **sys.dm_exec_requests**, and **sys.dm_exec_sessions**) are augmented with **pool_id** and/or **group_id** columns that reflect the Resource Governor configuration.

External Physical Memory Pressure

Look at the **Memory: Available [M, K]Bytes** performance counter. If the available memory amount is low, external memory pressure might be present. The exact value depends on many factors, such as the total amount of installed physical memory or the architecture (32-bit or 64-bit); however, you can start looking into this when the value drops below 50-100 MB. As with any troubleshooting strategy, keeping a baseline of a normally operating system provides you with a good reference value to look for.

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 the **Process: Working Set** performance counters 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**
  - **Modified Page List Bytes** counter
- If there's no external pressure, the **Process: Private Bytes** counter should be close to the size of the working set (**Process: Working Set**), which means that no memory is paged out.

Note that the corresponding performance counters do not count memory that is allocated through AWE mechanisms. Thus the information is insufficient if AWE is enabled or locked pages are used. 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 the 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 they only do so if AWE is enabled.

In the 64-bit version of SQL Server, if the "lock pages in memory" privilege is granted to the account running SQL Server 2008 or SQL Server 2005 and the edition (Enterprise or Developer) allows the use of locked pages, SQL Server allocates memory using the AWE mechanism. There is no need to set the **awe enabled** option explicitly. Memory allocated through the AWE API will be reflected in the output of the previous query.

In SQL Server 2008, you can obtain this information simply by querying the **sys.dm_os_process_memory** DMV.

The **physical_memory_in_use** column indicates total memory usage of the server including allocation through large page and AWE APIs. The **large_page_allocations_kb** and **locked_pages_allocations_kb** columns show the amount of memory allocated using large page and AWE APIs, respectively.

process_physical_memory_low = 1 indicates that the process is responding to physical memory low notification. A **memory_utilization_percentage** value below 100% under normal load may warrant investigation if the server is under external memory pressure.

Similarly, you can use the **sys.dm_os_sys_memory** DMV in SQL Server 2008 to assess the system state.

```
select
     total_physical_memory_kb / 1024 as phys_mem_mb,
     available_physical_memory_kb / 1024 as
avail_phys_mem_mb,
     system_cache_kb /1024 as sys_cache_mb,
     (kernel_paged_pool_kb+kernel_nonpaged_pool_kb) / 1024
          as kernel_pool_mb,
     total_page_file_kb / 1024 as total_page_file_mb,
     available_page_file_kb / 1024 as available_page_file_mb,
     system_memory_state_desc
from sys.dm_os_sys_memory
```

This query returns information about the state of the system (totals and available numbers) as well as the global state if the system detects low, high, or steady memory conditions. The "Available physical memory is low" indicator in the **system_memory_state_desc** column is another sign of external memory pressure that requires further investigation. Relieving external memory pressure by identifying and eliminating major physical memory consumers (if possible) and/or by adding more memory should generally resolve problems related to memory.

External Virtual Memory Pressure

You need to determine whether page file(s) have enough space to accommodate current memory allocations. To check this, look at the following counters: **Memory: Commit Limit**, **Paging File: %Usage**, **Paging File: %Usage Peak**. **Commit Limit** is the amount of virtual memory that can be committed without extending page file space.

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** is high, check the System Event log for events that could indicate 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).

In SQL Server 2008 you can assess the state of the page file limits from corresponding columns of the **sys.dm_os_sys_memory** DMV. **total_page_file_kb** matches the **Commit Limit** performance counter. **available_page_file_kb** represents available memory commitment. Note that the difference between these two values does not reflect actual page file usage. It equals to the current commit charge value for the system.

Internal Physical Memory Pressure

Because 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, take a look at the DBCC MEMORYSTATUS output. In the Buffer Pool section, look for the `Target` value. The following shows partial output of DBCC MEMORYSTATUS output on idle server.

```
Buffer Pool                                Value
----------------------------------------- -----------
Committed                                  11141
Target                                     242203
Database                                   5446
Dirty                                      94
In IO                                      0
Latched                                    0
Free                                       444
Stolen                                     5251
Reserved                                   0
Visible                                    242203
Stolen Potential                           733675
...
```

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 the Committed amount is above Target and you have ruled out external pressure, continue with your investigation of the largest memory consumers inside SQL Server.

Note that if the server is not loaded, Target is likely to exceed Committed and the amount reported by the **Process: Private Bytes** performance counter, which is normal.

If Target is low, but the server **Process: Private Bytes** is high, you 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, and SQLCLR, 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 multipage allocator interface.

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

```
-- amount of memory 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 multipage allocator by using the following query.

```
select
    type, sum(multi_pages_kb) as [KB]
from
    sys.dm_os_memory_clerks
where
    multi_pages_kb != 0
group by type
order by 2 desc
```

If you are seeing large amounts of memory allocated through the multipage allocator, check the server configuration and try to identify the components that consume most of the memory by using the previous SELECT statement. Baseline values for the memory consumption should give an idea if any particular component is consuming unexpectedly large amount of memory.

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 External Physical Memory Pressure earlier in this white paper 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, check for external virtual memory pressure, general operating system failures to allocate memory, or limitations of the SQL Server edition you are using. The value of `Target` cannot exceed the **max server memory** option setting.

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

```
Buffer Pool                                        Value
----------------------------------------           -----------
Committed                                          11141
Target                                             238145
Database                                           5446
Dirty                                              94
In IO                                              0
Latched                                            0
Free                                               377
Stolen                                             5318
Reserved                                           0
Visible                                            238145
Stolen Potential                                   733608
Limiting Factor                                    13
Last OOM Factor                                    0
Page Life Expectancy                               272532
```

A high percentage (greater than75-80%) 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 bBuffer 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'
```

In SQL Server 2008, summary information of the memory allocations per memory node (similar to the initial output of DBCC MEMORYSTATUS) can be obtained using the **sys.dm_os_memory_nodes** DMV.

This information can be used instead of running DBCC MEMORYSTATUS to quickly obtain summary memory usage. For a NUMA computer, there will be a line for each node, and for an SMP computer there will be a single line of output.

If there is a Dedicated Administrator Connection (DAC) node present on the edition of SQL Server, you will see an additional line of output for the **node_id**. **node_id**= 32 indicates the 32-bit version of SQL Server, and **node_id** = 64 indicates the 64-bit version of SQL Server.

Detailed information for each component can be obtained as follows. (This includes memory allocated from both within and outside of 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 because
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 bBuffer pPool): '
    + 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, because 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 under normal operation and comparing it with the time when errors happen can help you narrow down your investigation of the problem.

You can also check the following performance counters for signs of memory pressure (for more information, see SQL Server 2008 Books Online):

- SQL Server: **Buffer Manager** object
- High number of Checkpoint pages/sec
- High number of Lazy writes/sec

Insufficient memory and I/O overhead are usually related bottlenecks. For more information, see I/O Bottlenecks later 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.

In SQL Server 2008 and SQL Server 2005, internal implementation differs from SQL Server 2000 in its uniform caching framework. In order to remove 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`, the server is under internal or 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 the **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

A significant amount of diagnostic memory information can be obtained from the **sys.dm_os_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. You can find out about nonempty ring buffers along with their event count using the following query.

```
select
      ring_buffer_type
      , count(*) as [Event count]
from sys.dm_os_ring_buffers
group by ring_buffer_type
order by ring_buffer_type
```

RING_BUFFER_SCHEDULER_MONITOR

Information from this ring buffer can be used assess the general state of the system. System health records are stored with 1-minute intervals and may look like the following.

```
<Record id = "4560" type ="RING_BUFFER_SCHEDULER_MONITOR"
time ="419955168">
  <SchedulerMonitorEvent>
    <SystemHealth>
      <ProcessUtilization>1</ProcessUtilization>
      <SystemIdle>77</SystemIdle>
      <UserModeTime>13572087</UserModeTime>
      <KernelModeTime>156001</KernelModeTime>
      <PageFaults>291</PageFaults>
      <WorkingSetDelta>184320</WorkingSetDelta>
      <MemoryUtilization>100</MemoryUtilization>
    </SystemHealth>
  </SchedulerMonitorEvent>
```

```
</Record>
```

Using a query similar to the following example, you can see overall state of the server as far as there are SystemHealth records present in this ring buffer.

```
-- to correlate events, convert timestamps into time
-- note that the RDTSC counter IS affected by variable clock speeds of the
CPU
declare @ts_now bigint
select @ts_now = ms_ticks from sys.dm_os_sys_info

-- "decompose" the records
select record_id,
      dateadd(ms, -1 * (@ts_now - [timestamp]), GetDate()) as EventTime
      ,SQLProcessUtilization
      ,SystemIdle
      ,100 - SystemIdle - SQLProcessUtilization as OtherProcessUtilization
      ,UserModeTime
      ,KernelModeTime
      ,PageFaults
      ,WorkingSetDelta
      ,MemoryUtilPct
from (
      select
            record.value('(./Record/@id)[1]', 'int') as record_id,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/SystemIdl
e)[1]', 'int') as SystemIdle,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/ProcessUt
ilization)[1]', 'int') as SQLProcessUtilization,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/UserModeT
ime)[1]', 'int') as UserModeTime,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/KernelMod
eTime)[1]', 'int') as KernelModeTime,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/PageFault
s)[1]', 'int') as PageFaults,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/WorkingSe
tDelta)[1]', 'int') as WorkingSetDelta,

      record.value('(./Record/SchedulerMonitorEvent/SystemHealth/MemoryUti
lization)[1]', 'int') as MemoryUtilPct,
            timestamp
      from (
            select timestamp, convert(xml, record) as record
            from sys.dm_os_ring_buffers
            where ring_buffer_type = N'RING_BUFFER_SCHEDULER_MONITOR'
            and record like '%<SystemHealth>%') as x
      ) as y
order by record_id desc
```

Output of the previous query provides information about the CPU utilization for SQL Server and other processes as well as other useful diagnostic information about the server state. A query similar to this example is used in Management Data Warehouse data collection.

RING_BUFFER_RESOURCE_MONITOR

You can use information from resource monitor notifications to identify memory state changes. There is a record in this ring buffer for every memory state change. 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 the **sys.dm_os_ring_buffers** DMV.

In SQL Server 2008 a record may look like this.

```
<Record id="0" type="RING_BUFFER_RESOURCE_MONITOR" time="146278552">
  <ResourceMonitor>
    <Notification>RESOURCE_MEMPHYSICAL_HIGH</Notification>
    <IndicatorsProcess>0</IndicatorsProcess>
    <IndicatorsSystem>1</IndicatorsSystem>
    <NodeId>0</NodeId>
    <Effect type="APPLY_LOWPM" state="EFFECT_OFF" reversed="0">0</Effect>
    <Effect type="APPLY_HIGHPM" state="EFFECT_ON" reversed="0">0</Effect>
    <Effect type="REVERT_HIGHPM" state="EFFECT_OFF"
reversed="0">0</Effect>
  </ResourceMonitor>
  <MemoryNode id="0">
    <ReservedMemory>6282232</ReservedMemory>
    <CommittedMemory>38712</CommittedMemory>
    <SharedMemory>0</SharedMemory>
    <AWEMemory>8192</AWEMemory>
    <SinglePagesMemory>3208</SinglePagesMemory>
    <MultiplePagesMemory>21896</MultiplePagesMemory>
  </MemoryNode>
  <MemoryRecord>
    <MemoryUtilization>100</MemoryUtilization>
    <TotalPhysicalMemory>6222536</TotalPhysicalMemory>
    <AvailablePhysicalMemory>3044516</AvailablePhysicalMemory>
    <TotalPageFile>12665292</TotalPageFile>
    <AvailablePageFile>9388376</AvailablePageFile>
    <TotalVirtualAddressSpace>8589934464</TotalVirtualAddressSpace>

<AvailableVirtualAddressSpace>8583481552</AvailableVirtualAddressSpace>

<AvailableExtendedVirtualAddressSpace>0</AvailableExtendedVirtualAddressSp
ace>
  </MemoryRecord>
</Record>
```

From this record, you can deduce that the server received a high physical memory notification. You can also see the amounts of memory in kilobytes. Using a query similar to the one in the previous section, you can get time-stamped information about server memory state changes.

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 contains records indicating server out-of-memory conditions. A record may look like this.

```
Record id="0" type="RING_BUFFER_OOM" time="423802767">
  <OOM>
    <Action>FAIL_PAGE_ALLOCATION</Action>
    <Resources>1</Resources>
    <Task>0x000000000050FDC8</Task>
    <Pool>258</Pool>
  </OOM>
  <MemoryNode id="0">
    <ReservedMemory>6292472</ReservedMemory>
    <CommittedMemory>48096</CommittedMemory>
    <SharedMemory>0</SharedMemory>
    <AWEMemory>114688</AWEMemory>
    <SinglePagesMemory>28848</SinglePagesMemory>
    <MultiplePagesMemory>26368</MultiplePagesMemory>
  </MemoryNode>
  <MemoryRecord>
    <MemoryUtilization>100</MemoryUtilization>
    <TotalPhysicalMemory>6222536</TotalPhysicalMemory>
    <AvailablePhysicalMemory>2074616</AvailablePhysicalMemory>
    <TotalPageFile>12665292</TotalPageFile>
    <AvailablePageFile>8327120</AvailablePageFile>
    <TotalVirtualAddressSpace>8589934464</TotalVirtualAddressSpace>

<AvailableVirtualAddressSpace>8583411684</AvailableVirtualAddressSpace>

<AvailableExtendedVirtualAddressSpace>0</AvailableExtendedVirtualAddressSp
ace>
  </MemoryRecord>
  <Stack>
... (output truncated)
  </Stack>
</Record>
```

This record tells you which operation failed (commit, reserve, or page allocation), the amount of memory requested, in which Resource Governor resource pool this allocation failed, and additional server memory state information.

RING_BUFFER_MEMORY_BROKER and Internal Memory Pressure

This ring buffer contains records of memory notifications for each Resource Governor resource pool. 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 the pages to be reclaimed from caches and other components using them.

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

Internal memory pressure notifications ('Shrink') can be observed by querying the memory broker ring buffer.

RING_BUFFER_BUFFER_POOL

This ring buffer contains records indicating severe buffer pool failures, including buffer pool out-of-memory conditions.

This record tells you what failure occurred (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 queried by 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, your system is likely to be experiencing VAS pressure. SQL Server 2008 and SQL Server 2005 monitor and respond to VAS pressure. SQL Server 2000 does not actively monitor for VAS pressure; instead it reacts by clearing caches when an out-of-virtual-memory error occurs.

In SQL Server 2008, you can assess the state of the virtual memory low condition by using the **sys.dm_os_process_memory** DMV. The **process_virtual_memory_low** column bit indicates whether a low virtual memory condition has been detected.

General Troubleshooting Steps in Case of Memory Errors

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

1. Verify that the server is operating under external memory pressure. If external pressure is present, try resolving it first, and then see whether the problem or errors still exist.
2. Collect and compare performance counters as outlined in the previous sections.
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.
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 in resource pool '*pool_name*' to run this query.

Causes

This is 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 2008 and SQL Server 2005 and certain configuration settings (such as the **max server memory** option and Resource Governor configuration settings), users are more likely to see this error. Usually the transaction that failed is not the cause of this error. Check the **out_of_memory_count** column of the **sys.dm_resource_governor_resource_pools** DMV. If this count is localized to a particular resource pool, Resource Governor configuration is the most likely reason.

Troubleshooting

Regardless of whether the error is consistent and repeatable (that is, it stays in the same state) or random (that is, it appears at random times with different states), you should 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. When you see this error, the best place to start investigation is the error log. It should contain output that looks something like this.

```
2009-01-28 04:27:15.43 spid51       Failed allocate pages:
FAIL_PAGE_ALLOCATION 1
```

or

```
2009-01-28 04:27:15.43 spid51       Failed Virtual Allocate Bytes:
FAIL_VIRTUAL_RESERVE 65536
```

The possible failures are:

- FAIL_PAGE_ALLOCATION followed by the number of pages attempted to allocate.
- FAIL_VIRTUAL_RESERVE followed by the number of bytes attempted to reserve.
- FAIL_VIRTUAL_COMMIT followed by the number of bytes attempted to commit.

Usually the task that first encountered the out-of-memory error is not the task that caused the condition. Most likely it is a cumulative effect of multiple tasks running. For the very common case of a single page allocation failure, your investigation should take the global picture into account.

The next piece of information from error log is the memory status output. Depending on the failure, you should look for single page, multipage, virtual reserved or committed numbers for individual memory clerks. Identifying the biggest memory consumers is key to proceeding with investigation. You may find that the biggest consumers are of the following type:

- MEMORYCLERK_* means that the server configuration or workload requires so much memory to be allocated. The offending workload can sometimes be identified just by the memory clerks, but more often you will have to drill further into the memory objects associated with the clerks in order to find out what causes such memory consumption.
- CACHESTORE_*, USERSTORE_*, OBJECTSTORE_* are the types of caches. Big consumption by a cache may mean the following:
    - Memory is allocated out of the cache but is not inserted yet as an entry that can be evicted. This is very similar to the MEMORYCLERK case discussed earlier.
    - All cache entries are in use so they cannot be evicted. You can confirm this by looking at the **sys.dm_os_memory_cache_counters** DMV and comparing the **entries_count** and **entries_in_use_count** columns.

    - Most cache entries are not in use. This case most likely indicates a bug in the server.
- MEMORYCLERK_SQLQERESERVATIONS shows how much memory has been reserved by the query execution (QE) to run queries with sorts/joins.

The memory status output in the error log also shows which Resource Governor resource pool memory is exhausted. The memory brokers for every pool show the memory distribution between stolen (compilation), cached, and reserved (granted) memory. The numbers for the three brokers correspond to the three bullet points in the previous list. Unfortunately there is no way to find out how much memory is allocated for a pool from a given clerk or memory object. The **sys.dm_os_memory_cache_entries** DMV is extended to show the **pool_id** each entry is associated with.

Possible solutions include the following:

- Remove external memory pressure.
- Increase the **max server memory** setting, and then adjust the MIN_MEMORY_PERCENT and MAX_MEMORY_PERCENT settings for the resource pool.
- Free caches by using one of the following commands: DBCC FREESYSTEMCACHE, DBCC FREESESSIONCACHE, or DBCC FREEPROCCACHE.

If the problem reappears, reduce the 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 2008 and 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 competes for memory resources with other queries. If the query exceeds a predefined time-out (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

- Follow general troubleshooting steps to see whether the server memory consumption is affected in general.
- Check the workload. Verify the amounts of memory consumed by different components. (For more information, see Internal Physical Memory Pressure earlier in this paper.)

- Check the output of DBCC MEMORYSTATUS for the number of waiters at each gateway (this information will tell you whether 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
```

- Check for Resource Governor configuration.
- Reduce workload if possible.

8645 - A timeout occurred while waiting for memory resources to execute the query in resource pool '*pool_name*' (*pool_id*). 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 can consume significant amounts of memory during execution. Query memory requirements are significantly increased if a high degree of parallelism is enabled or if a query operates on a partitioned table with nonaligned indexes. A query that cannot access the memory resources it requires within the predefined time-out (by default, the time-out is 25 times the estimated query cost, the **sp_configure** 'query wait' amount if it is set, or the Resource Governor workload group setting `request_memory_grant_timeout_sec`) receives this error. Usually, the query that receives the error is not the one that is consuming the memory.

Troubleshooting

- Follow general steps to assess server memory condition.
- Identify problematic queries: Check to see whether a significant number of queries operate on partitioned tables, check to see whether they use nonaligned indexes, and check to see whether there are many queries involving joins and/or sorts.
- Check the **sp_configure** parameters **degree of parallelism** and **min memory per query**. Try reducing the degree of parallelism and verify that **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.
- Find out whether queries are waiting on RESOURCE_SEMAPHORE. For more information, see Blocking later in this paper.
- Check Resource Governor configuration.

8651 - Could not perform the operation because the requested memory grant was not available in resource pool '%ls' (%ld). Rerun the query, reduce the query load, or check resource governor configuration setting.

Causes

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

Troubleshooting

- Follow general memory error troubleshooting steps.
- Verify that the **sp_configure min memory per query** option setting is not too high.
- Check Resource Governor configuration settings.

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 storing intermediate results, sorting, and keeping row versions. 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 data files, including **tempdb**, are randomly accessed. So as a general rule, you should have log files on a physical disk that is separate from the 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 whether you have I/O bottleneck. After 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 do not complete due to time-outs.

You can use the following performance counters to identify I/O bottlenecks. Note that 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, 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. The following list shows ranges of possible values and what the ranges mean:

- 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. The guidelines for the **Avg. Disk Sec/Read** values apply here.

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, there is an I/O bottleneck.

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

When you use these 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 might 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 borderline 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 the I/O has already completed, the worker proceeds normally. Otherwise, it waits on PAGEIOLATCH_EX or PAGEIOLATCH_SH, depending upon the type of request. You can use the following DMV query 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
```

A sample output follows.

```
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
```

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 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** columns to find the physical disk the files are mapped to. The **io_pending_ms_ticks** values 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 see an I/O bottleneck, your first instinct might be to upgrade the I/O subsystem to meet the workload requirements. This will definitely help, but before you go out and invest money in hardware, examine the I/O bottleneck to see whether it is the result of poor configuration and/or query plans. We recommend you to follow the steps below in strict order.

1. **Configuration:** 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 the following counters to identify memory pressure:
   - Buffer Cache hit ratio
   - Page Life Expectancy
   - Checkpoint pages/sec
   - Lazywrites/sec

   For more information about memory pressure, see Memory Bottlenecks earlier in this paper.
2. **Query Plans**: Examine execution plans and see which plans lead to more I/O being consumed. 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 or requests are generating the most I/O. Note that we are not accounting for physical writes. This is okay if you consider how databases work. The DML and 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 either by checkpoint or by the SQL Server lazy writer. You can use a DMV query like the following to find the 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 enables 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.

   Here is an example of a query that joins two tables with a hash join.

```
create table t1 (c1 int primary key, c2 int, c3 char(8000))
create table t2  (C4 int, c5 char(8000))
go

--load the data
declare @i int
select @i = 0
while (@i < 6000)
begin
    insert into t1 values (@i, @i + 1000, 'hello')
   insert into t2 values (@i,'there')
   set @i = @i + 1
end
--now run the following query
```

```
select c1, c5
from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4
order by c2

Run another query so that there are two queries to look at for I/O
stats

select SUM(c1) from t1
```

These two queries are run in the single batch. Next, use the following DMV query to examine the queries that generate the most I/Os

```
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,
    (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(MAX),text)) * 2
                ELSE statement_end_offset
            END - statement_start_offset)/2)
     FROM sys.dm_exec_sql_text(sql_handle)) AS query_text,
      (SELECT query_plan from sys.dm_exec_query_plan(plan_handle)) as
query_plan
FROM sys.dm_exec_query_stats

   ORDER BY (total_logical_reads +
   total_logical_writes)/execution_count DESC
```

You can, of course, change this query to get different views on the data. For example, to generate the five requests that generate the 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. The output of the query is as follows.

```
avg_logical_reads    avg_logical_writes    avg_phys_reads
-----------------    ------------------    ---------------
16639                10                    1098
6023                 0                     0
execution_count      stmt_start_offset
---------------      -----------------
1                    0
1                    154

Query_text                                       Query_plan
```

```
-----------------------------------          -----------
select c1, c5  from t1 INNER HASH JOIN …     <link to query plan>
select SUM(c1) from t1                       <link to query plan>
```

The output tells you several important things. First, it identifies the queries that are generating the most I/Os. You can also look at the SQL text to see whether the query needs to be re-examined to reduce I/Os. Verify that the query plan is optimal. For example, a new index might be helpful. Second, the second query in the batch does not incur any physical I/Os because all the pages needed for table t1 are already in the buffer pool. Third, the execution count can be used to identify whether it is a one-off query or the one that is executed frequently and therefore needs to be looked into carefully.

3. **Data Compression:** Starting with SQL Server 2008, you can use the data compression feature to reduce the size of tables and indexes, thereby reducing the size of the whole database. The compression achieved depends on the schema and the data distribution. Typically, you can achieve 50-60% compression. We have seen up to 90% compression in some cases. What it means to you is that if you are able to compress you active data 50%, you have in effect reduced your I/O requirements by half. Data compression comes at the cost of additional CPU, which needs to be weighed in for your workload. Here are some general strategies.

   First, why isn't compressing the whole database blindly such a good idea? Well, to give you an extreme example, if you have a heavily used table T with 10 pages in a database with millions of pages, there is no benefit in compressing T. Even if SQL Server could compress 10 pages to 1 page, you hardly made a dent in the size of the database, but you did add some CPU overhead instead. In a real-life workload, the choices are not this obvious, but this example shows that you must look before you compress. Our recommendation is this: Before you compress an object (for example, a table index or a partition), look at its size, usage, and estimated compression savings by using the **sp_estimate_data_compression_savings** stored procedure.

   Let us look at each of these in some detail:

   - If the size of the object is much smaller than the overall size of the database, it does not buy you much.
   - If the object is used heavily both for DML and SELECT operations, you will incur additional CPU overhead that can impact your workload, especially if it makes it CPU bound. You can use **sys.dm_db_index _operational_stats** to find the usage pattern of objects to identify which tables, indexes, and partitions are being hit the most.
   - The compression savings are schema-dependent and data-dependent, and in fact, for some objects, the size after compression can be larger than before, or the space savings can be insignificant.

   If you have a partitioned table where data in some partitions is accessed infrequently, you may want to compress those partitions and associated indexes with page compression. This is a common scenario with partitioned tables where older partitions are referenced infrequently. For example, you might have a table in which sales data is partitioned by quarters across many years. Commonly the queries are run on the current quarter; data from other quarters is not referenced as frequently. So when the current quarter ends, you can change the compression setting for that quarter's partition.

For more information about data compression, see the SQL Server Storage Engine Blog (http://blogs.msdn.com/sqlserverstorageengine/archive/tags/Data+Compression/default.aspx) and SQL Server 2008 Books Online.

4. **Upgrading the I/O Subsystem**: If you have confirmed that SQL Server is configured correctly and examined the query plans and you are still experiencing I/O bottlenecks, the last option left is to upgrade your I/O subsystem to 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.

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. **tempdb** can become overloaded in terms of space available and excessive DDL and 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 earlier in this paper.
- 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.

Category	**Description**
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 the 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.
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 than it is to work to solve it later. You can use the **Free Space in tempdb** (KB) performance counter to monitor the amount of space **tempdb** is using. This counter tracks free space in **tempdb** in kilobytes. Administrators can use this counter to determine whether **tempdb** is running low on free space.

However, identifying how the different categories, as defined earlier, 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

Because 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 2008 provides a row versioning framework. Currently, the following features use the row versioning framework:

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

For more information about these features, see Row Versioning Resource Usage (http://msdn.microsoft.com/en-us/library/ms175492.aspx) in SQL Server 2008 Books Online.

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 stop 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 6,523 seconds.

```
transaction_id       transaction_sequence_num elapsed_time_seconds

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

8609                 3                        6523

20156                25                       783
```

Because the second transaction has been active for a relatively short period, you might be able to free up a significant amount of version store by stopping the first transaction. However, there is no way to estimate how much version space will be freed up by stopping this transaction. You may need to stop 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 with snapshot isolation or long-running queries with read-committed-snapshot isolation. 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 data generated per minute. However, there are some exceptions: Only differences are logged for updates; and a newly inserted data row is not versioned, but it might be logged, if it is a bulk-logged operation and the recovery mode is not 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, a long-running transaction could be preventing the version store cleanup.

Incidentally, before generating an out-of-**tempdb**-space error, SQL Server 2008 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, 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 table in tempdb earlier in this paper. If you notice that a huge amount of **tempdb** space is allocated, you should determine which session or tasks are consuming the space and then possibly take corrective action.

SQL Server 2008 provides two 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 tasks 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 sample output.

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

After 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 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** columns 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, 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 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. We recommend that you preallocate **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 numbers of temporary tables and table variables can cause contention on metadata. In SQL Server 2008, 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.
- 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.
- The temp object is not created by using dynamic SQL, such as: sp_executesql N'create table #t(a int)'.
- The temp object is created inside another object, such as a stored procedure, trigger, or user-defined function; or the temp object 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 smaller than 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 2008 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.

Because 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 511,232 pages, and each PFS page occurs after every 8,088 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 shows up in **sys.dm_os_waiting_tasks**. Because latch waits are transient, you should 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 whether 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 Performance Monitor 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 should look at your application and see whether 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 might 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-running or long-running queries can contribute to excessive resource consumption. They can be the consequence of blocked queries.

Excessive resource consumption is not restricted to CPU resources, but it 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. Because all indexes have to be maintained, this does not impact the performance of a query, but it 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 a 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 exclusive (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 noncompatible lock on an already-locked resource is made. Although 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 more information about concurrency control and blocking, see SQL Server 2008 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 and 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, a wait occurs if the resource has a queue of outstanding requests. DMVs can provide information for any sessions that are waiting on resources.

SQL Server 2008 provides detailed and consistent wait information, reporting approximately 125 wait types. 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**, which 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
```

This result shows that session 56 is blocked by session 53 and that session 56 has been waiting for a lock for 1,103,500 milliseconds:

- waiting_task_address: 0x022A8898
- session_id: 56
- exec_context_id: 0
- wait_duration_ms: 1103500
- wait_type: LCK_M_S
- resource_address: 0x03696820
- blocking_task_address: 0x022A8D48
- blocking_session_id: 53
- blocking_exec_context_id: NULL
- resource_description: ridlock fileid=1 pageid=143 dbid=9 id=lock3667d00 mode=X associatedObjectId=72057594038321152

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 and its output show 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
```

Locking Granularity and Lock Escalation

One of keys to understanding blocking is the transaction isolations and the locking granularity. Transaction isolation levels govern the duration for S mode locks but they don't impact the duration of exclusive (X) locks that are held for the duration of the transaction under all transaction isolation levels. The locking granularity determines whether the lock is to be acquired at row, level, page level, or table level. Clearly, the higher the locking granularity, the lower the concurrency. For example, if a transaction takes an exclusive (X) lock on the table to modify one row in a table, it blocks other transactions that want to read or modify completely different rows. On the other hand, the benefit of higher locking granularity is that SQL Server does not need to acquire as many locks, which saves both memory and the CPU cost of acquiring and releasing the locks. The locking granularity is determined by SQL Server using a heuristic model that works pretty well, but there may be cases where does not behave as expected. For cases like this, user can use **sp_tableoption** or ALTER INDEX DDL to control the locking granularity on the object or by providing locking hints.

There is another interesting twist with locking. SQL Server can escalate the lock on a table if it determines at run time that the number of locks acquired on an object in a statement has exceeded a threshold. A lock escalation is triggered when any of the following conditions is true:

- The number of locks held (as opposed to those that are acquired and then released; for example, when one or more rows are read under the read committed isolation level) by a statement on an index or a heap within a statement exceeds the threshold, which is set to approximately 5000 by default. These locks include intent locks as well. Note that the lock escalation will not trigger if:
    - The transaction acquires 2,500 locks each on two indexes or heaps in a single statement.
    - The transaction acquires 2,500 locks on the nonclustered index and 2,500 locks on the corresponding base table in a single statement.

    - The same heap or index is referenced more than one time in a statement; the locks on each instance of those are counted separately. So for example, in the case of a self-join on a table t1, if each instance has 3,000 locks within the statement, lock escalation is not triggered.

- The memory taken by lock resources is greater than 40% of the non-AWE (32-bit) or regular (64-bit) enabled memory if the locks configuration option is set to 0, the default value. In this case, the lock memory is allocated dynamically as needed.

When the lock escalation is triggered, SQL Server attempts to escalate the lock to table level, but the attempt may fail if there are conflicting locks. So for example, if the SH locks need to be escalated to the table level and there are concurrent exclusive (X) locks on one or more rows or pages of the target table, the lock escalation attempt fails. However, SQL Server periodically, for every 1,250 (approximately) new locks acquired by the lock owner (that is, the transaction that initiates the lock), attempts to escalate the lock. If the lock escalation succeeds, the SQL Server releases the lower-granularity locks, and the associated lock memory, on the index or the heap. Because at the time of lock escalation, there cannot be any conflicting access, a successful lock escalation can potentially lead to blocking of future concurrent access to the index or the heap by transactions in conflicting lock modes. Therefore, lock escalation is not always a good idea for all applications.

Disabling Lock Escalation

SQL Server provides the following trace flags to disable lock escalation:

- TraceFlag-1211: It disables lock escalation at the current threshold (5000) on a per-index or per-heap per-statement basis. If this trace flag is in effect, the locks are never escalated. This trace flag also instructs SQL Server to ignore the memory acquired by the lock manager up to a maximum statically allocated lock memory or 60% of non-AWE (32-bit) or regular (64-bit) dynamically allocated memory. At this time an out-of-lock-memory error is generated. This can potentially be damaging, because an application can exhaust SQL Server memory by acquiring large number of locks. This, in the worst case, can stall the server or degrade its performance to an unacceptable level. For these reasons, you must exercise caution when you use this trace flag.
- TraceFlag-1224: This trace flag is similar to trace flag 1211 with one key difference. It enables lock escalation if the lock manager acquires 40% of the statically allocated memory or 40% of the non-AWE (32-bit) or regular (64-bit) dynamically allocated memory. Additionally, if this memory cannot be allocated because other components are taking up more memory, the lock escalation can be triggered earlier. SQL Server generates an out-of-memory error when memory allocated to the lock manager exceeds the statically allocated memory or 60% of non-AWE (32-bit) or regular (64-bit) dynamically allocated memory.

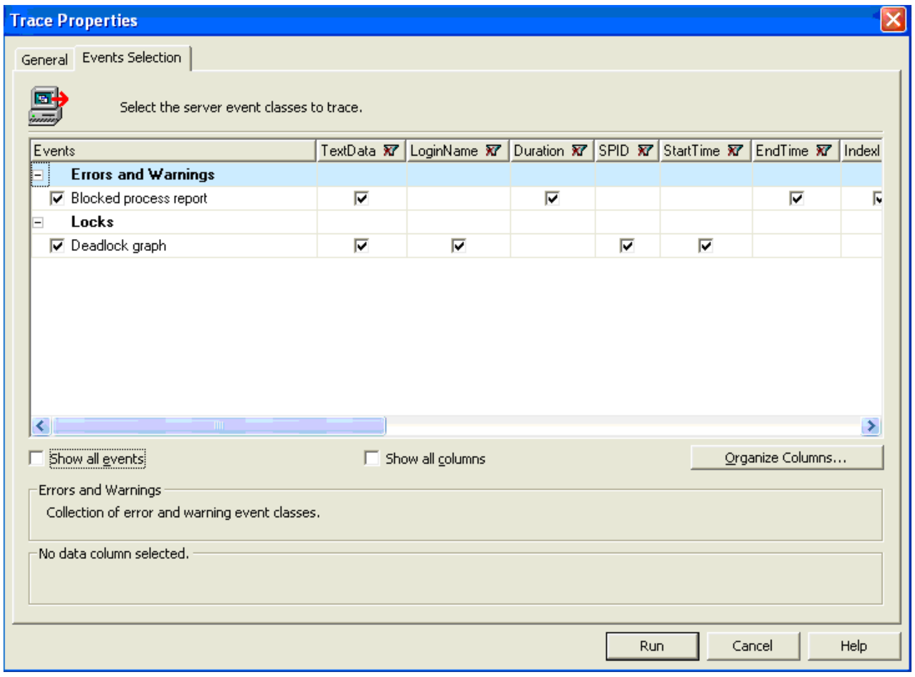
If both trace flags (1211 and 1224) are set at the same time, trace flag 1211 takes precedence. You can use the DBCC TRACESTATUS (-1) command to find the status of all trace flags enabled in SQL Server. The limitation with these trace flags is that they are coarse in granularity (that is, they are at the SQL Server instance level). SQL Server 2008 addresses this by providing an option at the table level to enable or disable lock escalation. Secondly, SQL Server 2008 provides lock escalation at the partition level so that lock escalation due to DML activities in one partition does not impact access to other partitions.

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 exceeds a threshold, it may impact the response time. To identify long-running blocking, you can use the 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.
3. After the blocked process threshold is established, capture the trace event. The trace events of blocking events that exceed the user configured threshold can be captured with SQL Trace or SQL Server Profiler.
4. If you are using SQL Trace, use **sp_trace_setevent** and **event_id**=137.
5. If you are using SQL Server Profiler, select the Blocked process report event class (under the Errors and Warnings object). See Figure 1.



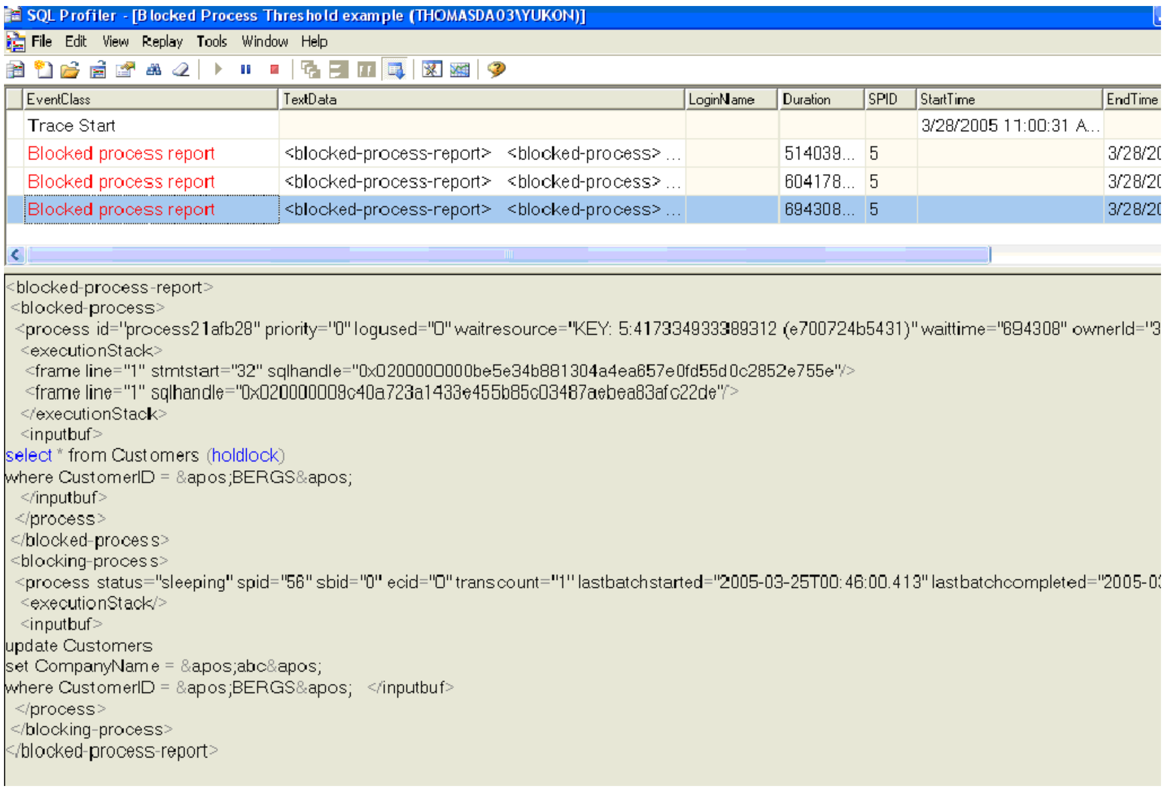
Events	TextData	LoginName	Duration	SPID	StartTime	EndTime	IndexI
Errors and Warnings							
✓ Blocked process report	✓		✓			✓	✓
Locks							
✓ Deadlock graph	✓	✓		✓	✓		

Errors and Warnings
Collection of error and warning event classes.

No data column selected.

Figure 1: Tracing long blocks and deadlocks

Note This is a lightweight trace, because 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 three trace events. See Figure 2.


SQL Profiler - [Blocked Process Threshold example (THOMASDA03\YUKON)]

EventClass	TextData	LoginName	Duration	SPID	StartTime	EndTime
Trace Start					3/28/2005 11:00:31 A...	
Blocked process report	<blocked-process-report> <blocked-process> ...		514039...	5		3/28/20
Blocked process report	<blocked-process-report> <blocked-process> ...		604178...	5		3/28/20
Blocked process report	<blocked-process-report> <blocked-process> ...		694308...	5		3/28/20

```
<blocked-process-report>
 <blocked-process>
  <process id="process21afb28" priority="0" logused="0" waitresource="KEY: 5:417334933389312 (e700724b5431)" waittime="694308" ownerId="3
   <executionStack>
    <frame line="1" stmtstart="32" sqlhandle="0x0200000000be5e34b881304a4ea657e0fd55d0c2852e755e"/>
    <frame line="1" sqlhandle="0x020000009c40a723a1433e455b85c03487aebea83afc22de"/>
   </executionStack>
   <inputbuf>
select * from Customers (holdlock)
where CustomerID = &apos;BERGS&apos;
   </inputbuf>
  </process>
 </blocked-process>
 <blocking-process>
  <process status="sleeping" spid="56" sbid="0" ecid="0" transcount="1" lastbatchstarted="2005-03-25T00:46:00.413" lastbatchcompleted="2005-0
   <executionStack/>
   <inputbuf>
update Customers
set CompanyName = &apos;abc&apos;
where CustomerID = &apos;BERGS&apos;   </inputbuf>
  </process>
 </blocking-process>
</blocked-process-report>
```

Figure 2: 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.

Blocking per Object with sys.dm_db_index_operational_stats

The 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 include a history of accesses, locks (**row_lock_count**), blocks (**process_virtual_memory_low**), and waits (**row_lock_wait_in_ms**) for a given index or table.

The type of information available through this DMV includes:

- The count of locks held, for example, row or page.
- The count of blocks or waits, for example, row or page.
- The duration of blocks or waits, for example, row or page.
- The count of page latches waits. The 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.
- The duration of **page_io_latch_wait**: An I/O latch occurs when a user requests a page that is not in the buffer pool. A slow or overworked I/O subsystem can sometimes experience high PageIOlatch waits that are actually I/O issues. These issues can be compounded by cache flushes or missing indexes.
- The duration of page latch waits.

Other types of information are also kept for access to indexes:

- Types of access, for example, range or singleton lookups.
- Inserts, updates, and 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. These pages represent empty pages that were deallocated because rows were deleted.
- Index maintenance. Pages merged above the leaf level are empty pages that were deallocated because to rows were deleted at the 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 that represents 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 so that you can query it further.

Overall Performance Effect of Blocking Using Waits

SQL Server 2008 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.

Session-Level Wait Statistics

To identify session-level or statement-level wait statistics, which was not possible in previous versions of SQL Server, Extended Events is an ideal tool. It is scalable and able to trace wait statistics during run time.

This example Extended Events session traces all waits, including those that are both internal and external to SQL Server, for session id 54.

```
-- sqlserver.session_id is the ID of the target session you want to trace.
I am using 54 in the example. Replace it accordingly
-- make sure C:\xevent folder exists
create event session session_waits on server
add event sqlos.wait_info
      (action (sqlserver.sql_text, sqlserver.plan_handle,
sqlserver.tsql_stack)
WHERE sqlserver.session_id=54 and duration>0)
, add event sqlos.wait_info_external
      (action (sqlserver.sql_text, sqlserver.plan_handle,
sqlserver.tsql_stack)
WHERE sqlserver.session_id=54 and duration>0)
add target package0.asynchronous_file_target
      (SET filename=N'C:\xevent\wait_stats.xel',
metadatafile=N'C:\xevent\wait_stats.xem');

alter event session session_waits on server state = start;
go

-- wait for monitored workload in target session (54 in this example) to
finish.
```

To read the results from the output file, run the following queries.

```
alter event session session_waits on server state = stop
drop event session session_waits on server

select
CONVERT(xml, event_data).value('(/event/data/text)[1]','nvarchar(50)') as
'wait_type',
CONVERT(xml, event_data).value('(/event/data/value)[3]','int') as
'duration',
CONVERT(xml, event_data).value('(/event/data/value)[6]','int') as
'signal_duration'
into #eventdata
from sys.fn_xe_file_target_read_file
(N'C:\xevent\wait_stats*.xel', N'C:\xevent\wait_stats*.xem', null, null)
-- save to temp table, #eventdata
select wait_type, SUM(duration) as 'total_duration', SUM(signal_duration)
as 'total_signal_duration'
from #eventdata
group by wait_type

drop table #eventdata
go
```

Sample output is shown here.

```
wait_type                                        total_duration       total_signal_duration
------------------------------------------------ -------------- ---------------------
NETWORK_IO                                        233                 0
PREEMPTIVE_OS_WAITFORSINGLEOBJECT                 231                 576
WAITFOR                                           7000                0
PAGEIOLATCH_UP                                    624                 0
PAGELATCH_UP                                      2320                45
PAGELATCH_SH                                      45                  10
WRITELOG                                          30                  0
```

Monitoring Index Usage

Another aspect of query performance is related to DML queries and queries that delete, insert, and modify data. The more indexes defined on a specific table, the more resources 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 indexes that never get used.

SQL Server 2008 provides the **sys.dm_db_index_usage_stats** DMV, which 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, and UPDATE operations. *System type* operations are commands like DBCC statements, 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**. **index_id**=0 represents a heap table, **index_id**=1 represents a clustered index, and **index_id**>1 represents a nonclustered index.

Over days of run time of an application against a database, the list of indexes getting accessed in **sys.dm_db_index_usage_stats** grows.

The rules and definitions for seek, scan, and lookup work as follows in SQL Server 2008:

- Seek: Indicates how many times 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 you can expect most of the hits against an index to be accumulated in this category.

- Scan: Indicates how many times the data layer of the table gets used for retrieval without using one of the index B-trees. For 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 was used to look up data that was identified by seeking through a nonclustered index that is also defined on that table. This describes the scenario known as *bookmark lookup* in SQL Server 2000 or earlier. 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 increments 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, and if the number of user seeks against one particular nonclustered index is very high as well, you should consider making the nonclustered index with the high count the clustered index.

The following DMV query can be used to obtain useful information about 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
```

You 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, because we see 123 lookups on the clustered index.

The most interesting columns of **sys.dm_db_index_usage_stats** to look at are the user type columns, including **user_seeks** and **user_scans**. System usage columns like **system_seeks** 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 that 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 DMV 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 were not used for months or during periods such as quarter-end reporting or fiscal-year reporting, you could eventually delete those indexes from the database.

Extended Events

SQL Server 2008 introduced Extended Events as a general event handling system for the server. Essentially, Extended Events is an infrastructure that supports event generation and processing for interesting execution points. Event generation can be done synchronously or asynchronously. Performing event generation synchronously ensures no data loss, Similarly, event processing can be done both ways. Asynchronous event processing ensures scalability and has the least performance overhead on the server system. Extended Events is designed to be a foundation that users can configure to monitor and capture different types of data, including performance data. It's a flexible and powerful way to provide a low granular level of information about the server system.

From a high level, Extended Events is implemented in packages, which are containers of objects like events, targets, and actions. Multiple packages can be implemented in a single module, which can be an executable or a dynamic linked library (DLL). The following figure shows the relationships among various objects, a package, and a module.

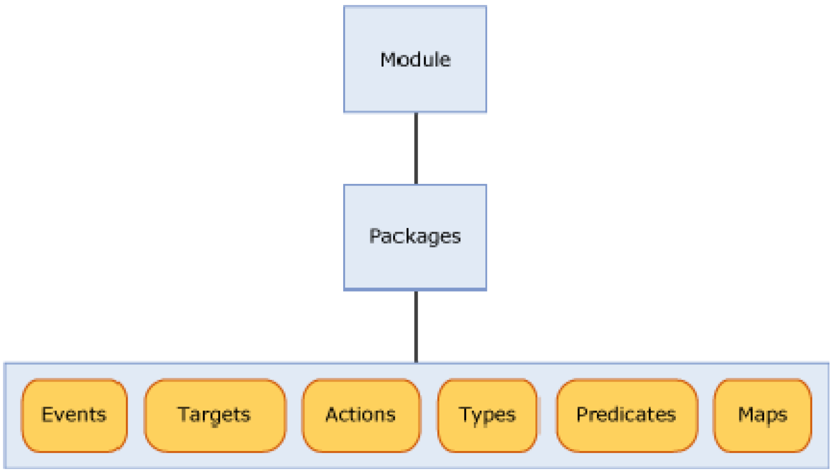


Figure 4. Extended Events relationships

Events – events represent the execution points of the process, in this case SQL Server. Events carry relevant information themselves, and they can also be fired upon to trigger actions.

Targets – targets consume events and indicate where output is located, such as a file, ring_buffer, or a bucket with aggregation. Targets can process events synchronously or asynchronously.

Actions – actions are performed synchronously in association with bound events. They can be used to accomplish certain tasks or simply provide more information relevant to the events.

Types –types determine the type of the package objects. They are based on the length and pattern of collected data.

Predicates – predicates are logical rules used to filter captured events. They help reduce the volume of captured data and tailor down the output for analysis.

Maps – maps are tables that map internal object values to user-friendly descriptions.

For more information about these objects, see SQL Server Extended Events Packages (http://msdn.microsoft.com/en-us/library/bb677278.aspx) in SQL Server 2008 Books Online.

Create an Extended Events Session

Using Extended Events packages, you can create sessions to host the Extended Events engine and perform actions against various targets. For example, the following code creates a session that traces event of database log auto growth.

```
-- make sure the C:\xevent folder exists
create event session log_growth on server
     ADD EVENT sqlserver.databases_log_growth
```

```
            (action (sqlserver.database_id))
add target package0.asynchronous_file_target
      (SET filename=N'C:\xevent\log_growth.xel',
metadatafile=N'C:\xevent\log_growth.xem');
```

In this definition of the session, the event is "database_log_growth", which belongs to the package "sqlserver". The action is to collect **database id**, which indicates which database experienced log growth. `target` is an asynchronous file, log_growth.xel (the metadata file log_growth.xem is an optional file that describes events data in log_growth.xel). To start this session, run the following.

```
alter event session log_growth on server state = start;
```

After the event of log auto growth occurs, you can stop the session and retrieve the output with code similar to the following.

```
alter event session log_growth on server state = stop
select *
from sys.fn_xe_file_target_read_file
(N'C:\xevent\log_growth*.xel', N'C:\xevent\log_growth*.xem',
null, null)
```

Other Common Tasks for Extended Events Sessions

To view all created sessions, use this query.

```
select * from sys.server_event_sessions
```

To view all active(running) sessions, use this query.

```
select * from sys.dm_xe_sessions
```

To drop a session, use this query.

```
drop event session <session_name> on server
```

Package and Object Information

As you can tell from the previous example, to create a meaningful extended event session, you need to know which event, action, target, predicate, and so forth to add. SQL Server 2008 ships with predefined packages with related object information. To view all registered packages, use the following example.

```
select * from sys.dm_xe_packages
```

To view all available events, actions, targets, and other object information for the packages, use the following example.

```
select p.name as package, xo.name as object_name,
xo.description, xo.object_type
from sys.dm_xe_objects xo
join sys.dm_xe_packages p
on xo.package_guid = p.guid
```

For reference, SQL Server 2008 includes the following DMVs (run-time information) for the Extended Events engine:

- **sys.dm_xe_packages** --Lists all the packages registered with the extended events engine
- **sys.dm_xe_objects** --Returns a row for each object that is exposed by an event package
- **sys.dm_xe_object_columns** --Returns the schema information for all the objects
- **sys.dm_xe_map_values** --Returns a mapping of internal numeric keys to human-readable text
- **sys.dm_xe_sessions** --Returns information about an active extended events session
- **sys.dm_xe_session_targets** --Returns information about session targets
- **sys.dm_xe_session_events** --Returns information about session events
- **sys.dm_xe_session_event_actions** --Returns information about event session actions
- **sys.dm_xe_session_object_columns** --Shows the configuration values for objects that are bound to a session

SQL Server 2008 also includes the following system catalog views (metadata) for event sessions:

- **sys.server_event_sessions** --Lists all the event session definitions
- **sys.server_event_session_events** --Returns a row for each event in an event session
- **sys.server_event_session_actions** --Returns a row for each action on each event of an event session
- **sys.server_event_session_targets** --Returns a row for each event target for an event session
- **sys.server_event_session_fields** --Returns a row for each customizable column that was explicitly set on events and targets

System_health Event Session

In addition, SQL Server 2008 includes a default Extended Events session, “system_health”, which is started automatically when the SQL Server service starts. In fact, you probably already have an Extended Events session running at this moment as part of the default installation of SQL Server 2008. If you query against **sys.server_event_sessions**, you should be able to see it.

```
select event_session_id, name, startup_state from
sys.server_event_sessions
```

This query returns the following results.

```
Event_session_id   name              startup_state
65536              system_health     1
```

For startup_state, a value of 1 indicates that the session is configured to start when the SQL Server service is started. To find out whether the session is indeed up and running, you can query against sys.dm_xe_sessions.

```
select name, create_time from sys.dm_xe_sessions where name =
'system_health'
```

You should be able to see an entry here indicating it's actively running.

```
Name             create_time
system_health    2009-03-13 13:58:34.690
```

To find out the event consumer (that is, the target), you can run the following query.

```
SELECT target_name
FROM sys.dm_xe_sessions x INNER JOIN
sys.dm_xe_session_targets xt
ON x.address = xt.event_session_address
WHERE x.name = 'system_health'
```

This query returns the following.

```
target_name
ring_buffer
```

The system_health session is no different from any user-defined session, like the log_growth session created earlier in this white paper. So you can stop and/or drop the system_health session just like any user-defined session. If you need to re-create the session, you can use the installed script from \\..\mssql\install\u_tables.sql (by default, this is installed to C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\Install). By looking at the session definition from the script, you can see that it traces:

- Reported errors with a severity of 20 or above.
- Nonyielding scheduler events.
- Deadlock.
- Certain waits that are longer than 15 seconds.

To test the system_health session, you can look at its reported_error events. For example, if a severity 20 error message is raised, Based on the default definition of system_health, it is traced.

```
--IMPORTANT: Do not run  run this on a production SQL Server. This would stop the current connection
--This would raise a severity 20 state 1 user error and then disconnect
raiserror('user test error message', 20, 1) with log
```

Let's look at the output.

```
SELECT
xdata.value('(/RingBufferTarget/event/action/value)[2]','int'
) AS 'session id',
xdata.value('(/RingBufferTarget/event/action/value)[3]','nvar
char(255)') AS 'statement'
FROM
(SELECT CONVERT(xml, xt.target_data) AS xdata
FROM sys.dm_xe_sessions x INNER JOIN
sys.dm_xe_session_targets xt
ON x.address = xt.event_session_address
WHERE x.name = 'system_health') AS xe

session id  statement
----------- -------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
---------------------------------------------
54          raiserror('user test error message', 20, 1) with log
```

To customize the way system_health session handles errors, you can change its definition or create a new session. Here is an example of a new session to capture **sql_batch**, **tsql_stack**, and **client_app_name** when user-defined error 50001 is raised.

```
create event session session_error on server
      ADD EVENT sqlserver.error_reported
            (action (sqlserver.sql_text, sqlserver.tsql_stack,
sqlserver.client_app_name)
            where error = 50001)
ADD target package0.ring_buffer
with (max_dispatch_latency = 1 seconds)
```

To summarize: Extended Events is a robust tool that can help troubleshoot various performance problems. These problems range from run-time errors and resource contention to waits and hot spots. Extended Events also helps troubleshoot problems like blocking and deadlocking. Thanks to its light performance impact and flexible configuration, it’s an ideal tool to monitor and gather performance data on a live production environment.

Data Collector and the MDW

SQL Server 2008 introduced a new performance monitoring tool called data the data collector. The data collector stores data in the management data warehouse (MDW). There are five different collector types: T-SQL Query, SQL Trace, Performance Counters, and Query Activity. Out of the box, SQL Server 2008 provides the following system data collection definitions:

- Disk Usage. Collects local disk usage information for all the databases of the SQL Server instance. This information can help you determine space usage and space requirements for disk capacity planning.
- Server Activity. Collects SQL Server instance-level resource usage information like CPU, memory, and I/O. This information can help you monitor short-term to long-term resource usage trends and identify potential resource bottlenecks on the system. It can also be used for resource capacity planning.
- Query Statistics. Collects individual statement-level query statistics, including query text and query plans. This information can help you identify top resource consuming queries for performance tuning.

The data collector is implemented as SQL Server Information Services (SSIS) packages. These packages can be configured to run manually, continuously, or scheduled as SQL Server Agent jobs to periodically collect and upload data to a central database referred to as the management data warehouse (sometimes known as the MDW). The MDW is simply a database serving the purpose of storing the collected data for viewing and reporting. The following figure shows the architecture of the feature.

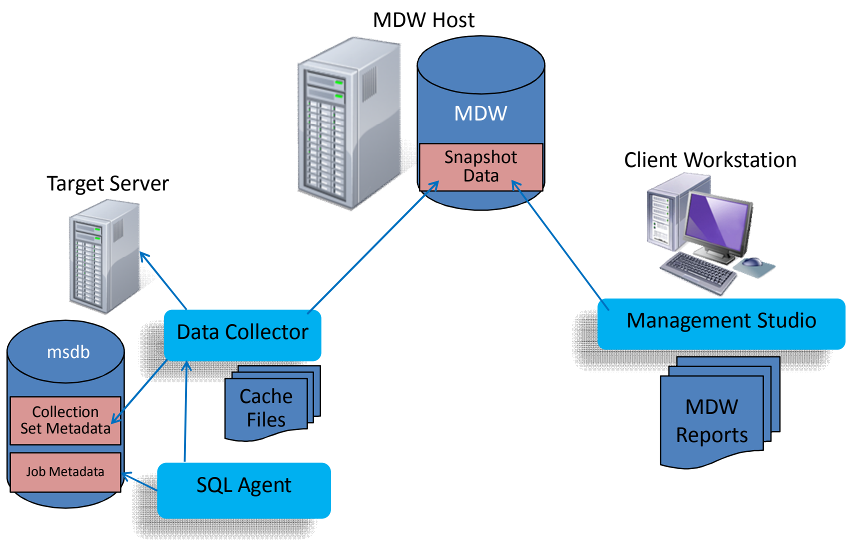


Figure 4. Data collection architecture

A single MDW database can serve as the central repository for data collectors running on multiple target SQL Server instances. The data collectors are configured on each target server, and they collect and upload data to the MDW database, which can be on a remote server. Between the time that the data is captured and the time it is uploaded, the data collector can write temporary data into cache files on the target server. The collection sets are normally run as SQL Server Agent jobs, so metadata about collection frequency, what items to collect, and so forth is stored in the **msdb** database. The system collection sets have predefined reports, which are accessed through SQL Server Management Studio and used to visualize the captured data.

You can change the schedule of collection sets if they are configured as scheduled jobs. You can also specify how long to keep the collected data and where to store cached data prior to upload. For example, by default, the Server Activity collection set collects data every minute and uploads data every 15 minutes. Depending on the time frame of your target window, you might choose to decrease the collection frequency to 5 minutes, which will capture only one-fifth of the data (mostly performance counters).

The definition of system collection sets cannot be modified. However, you can define your own collection sets and store that information in the MDW (tables will be created in the custom_snapshots schema) and define your own custom reports for this data.

Customized Data Collection

As mentioned earlier, the MDW as delivered with SQL Server 2008 can be used as a framework to build customized data collection for performance monitoring. We'll go through an example below to monitor a few Performance Monitor counters. The full working script is in Appendix B.

1. Use the Configure Management Data Warehouse wizard to configure a database as the MDW. (In SQL Server Management Studio, under **Management**, right-click **Data Collection**, and then click **Configure Management Data Warehouse**.)
2. Create data collection sets and add collection items.

```
Use msdb
go

Declare @collection_set_id_1 int
Declare @collection_set_uid_2 uniqueidentifier
EXEC [dbo].[sp_syscollector_create_collection_set]
   @name=N'Disk Performance and SQL CPU',
   @collection_mode=1,
   @description=N'Collects logical disk performance counters and SQL
Process CPU',
   @target=N'',
   @logging_level=0,
   @days_until_expiration=7,
   @proxy_name=N'',
   @schedule_name=N'CollectorSchedule_Every_5min',
   @collection_set_id=@collection_set_id_1 OUTPUT,
   @collection_set_uid=@collection_set_uid_2 OUTPUT
Select collection_set_id_1=@collection_set_id_1,
collection_set_uid_2=@collection_set_uid_2

Declare @collector_type_uid_3 uniqueidentifier
Select @collector_type_uid_3 = collector_type_uid From
[dbo].[syscollector_collector_types] Where name = N'Performance
Counters Collector Type';
Declare @collection_item_id_4 int
EXEC [dbo].[sp_syscollector_create_collection_item]
@name=N'Logical Disk Collection and SQL Server CPU',
@parameters=N'<ns:PerformanceCountersCollector
xmlns:ns="DataCollectorType">
   <PerformanceCounters Objects="LogicalDisk"
         Counters="Avg. Disk Bytes/Read"
         Instances="*" />
   <PerformanceCounters Objects="LogicalDisk"
         Counters="Avg. Disk Bytes/Write"
         Instances="*" />
   <PerformanceCounters Objects="LogicalDisk"
         Counters="Avg. Disk sec/Read"
         Instances="*" />
   <PerformanceCounters Objects="LogicalDisk"
         Counters="Avg. Disk sec/Write"
         Instances="*" />
   <PerformanceCounters Objects="LogicalDisk"
         Counters="Disk Read Bytes/sec"
         Instances="*" />
   <PerformanceCounters Objects="LogicalDisk"
         Counters="Disk Write Bytes/sec"
         Instances="*" />
   <PerformanceCounters Objects="Process"
         Counters="% Privileged Time"
```

```
            Instances="sqlservr" />
      <PerformanceCounters Objects="Process"
            Counters="% Processor Time"
            Instances="sqlservr" />
   </ns:PerformanceCountersCollector>',
   @collection_item_id=@collection_item_id_4 OUTPUT,
   @frequency=5,
   @collection_set_id=@collection_set_id_1,
   @collector_type_uid=@collector_type_uid_3
   Select @collection_item_id_4
   go
```

3. Start this data collection.
4.

```
EXEC sp_syscollector_start_collection_set @collection_set_id =
<collection_set_id_1>
-- replace <collection_set_id_1> with value from above
```

5. Read the output.

   Using the database that was configured as a data warehouse in step 1, run following query to get Performance Monitor counter values.

```
select spci.path as 'Counter Path', spci.object_name as 'Object
Name',
spci.counter_name as 'counter Name', spci.instance_name,
spcv.formatted_value as 'Formatted Value',
spcv.collection_time as 'Collection Time',
csii.instance_name as 'SQL Server Instance'
from snapshots.performance_counter_values spcv,
snapshots.performance_counter_instances spci,
msdb.dbo.syscollector_collection_sets_internal scsi,
core.source_info_internal csii,
core.snapshots_internal csi
where spcv.performance_counter_instance_id =
spci.performance_counter_id and
scsi.collection_set_uid=csii.collection_set_uid and
csii.source_id = csi.source_id and csi.snapshot_id=spcv.snapshot_id
and
scsi.name = 'Disk Performance and SQL CPU'
   order by spcv.collection_time desc
```

For more information about the data collector and the MDW, including how to set them up, see Introducing the Data Collector (http://msdn.microsoft.com/en-us/library/bb677248.aspx) in SQL Server 2008 Books Online.

Conclusion

For more information:

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

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.
Send feedback.

Appendix A: DBCC MEMORYSTATUS Description

Some types of information are primarily available through use of the DBCC MEMORYSTATUS command. However, some of this information is also available through the use of DMVs.

For more information about the DBCC MEMORYSTATUS command as used in SQL Server 2005, see Knowledge Base article 907877, How to use the DBCC MEMORYSTATUS command to monitor memory usage on SQL Server 2005 (http://support.microsoft.com/?id=907877).

Appendix B: MDW Data Collection

This appendix provides a sample script that creates a customized data collection.

```
use msdb;
Begin Transaction
Begin Try
Declare @collection_set_id_1 int
Declare @collection_set_uid_2 uniqueidentifier
EXEC [dbo].[sp_syscollector_create_collection_set]
      @name=N'Disk Performance and SQL CPU',
      @collection_mode=1,
      @description=N'Collects logical disk performance counters and SQL
Process CPU',
      @target=N'',
      @logging_level=0,
      @days_until_expiration=7,
      @proxy_name=N'',
      @schedule_name=N'CollectorSchedule_Every_5min',
      @collection_set_id=@collection_set_id_1 OUTPUT,
      @collection_set_uid=@collection_set_uid_2 OUTPUT
Select @collection_set_id_1, @collection_set_uid_2

Declare @collector_type_uid_3 uniqueidentifier
Select @collector_type_uid_3 = collector_type_uid From
[dbo].[syscollector_collector_types] Where name = N'Performance Counters
Collector Type';
Declare @collection_item_id_4 int
EXEC [dbo].[sp_syscollector_create_collection_item]
@name=N'Logical Disk Collection and SQL Server CPU',
@parameters=N'<ns:PerformanceCountersCollector
xmlns:ns="DataCollectorType">
      <PerformanceCounters Objects="LogicalDisk"
            Counters="Avg. Disk Bytes/Read"
            Instances="*" />
      <PerformanceCounters Objects="LogicalDisk"
            Counters="Avg. Disk Bytes/Write"
            Instances="*" />
      <PerformanceCounters Objects="LogicalDisk"
            Counters="Avg. Disk sec/Read"
            Instances="*" />
      <PerformanceCounters Objects="LogicalDisk"
            Counters="Avg. Disk sec/Write"
            Instances="*" />
      <PerformanceCounters Objects="LogicalDisk"
            Counters="Disk Read Bytes/sec"
            Instances="*" />
      <PerformanceCounters Objects="LogicalDisk"
            Counters="Disk Write Bytes/sec"
            Instances="*" />
      <PerformanceCounters Objects="Process"
            Counters="% Privileged Time"
            Instances="sqlservr" />
      <PerformanceCounters Objects="Process"
            Counters="% Processor Time"
            Instances="sqlservr" />
</ns:PerformanceCountersCollector>',
@collection_item_id=@collection_item_id_4 OUTPUT,
@frequency=5,
@collection_set_id=@collection_set_id_1,
@collector_type_uid=@collector_type_uid_3
Select @collection_item_id_4

Commit Transaction;
End Try
Begin Catch
Rollback Transaction;
DECLARE @ErrorMessage NVARCHAR(4000);
DECLARE @ErrorSeverity INT;
DECLARE @ErrorState INT;
DECLARE @ErrorNumber INT;
DECLARE @ErrorLine INT;
DECLARE @ErrorProcedure NVARCHAR(200);
SELECT @ErrorLine = ERROR_LINE(),
       @ErrorSeverity = ERROR_SEVERITY(),
       @ErrorState = ERROR_STATE(),
       @ErrorNumber = ERROR_NUMBER(),
       @ErrorMessage = ERROR_MESSAGE(),
       @ErrorProcedure = ISNULL(ERROR_PROCEDURE(), '-');
RAISERROR (14684, @ErrorSeverity, 1 , @ErrorNumber, @ErrorSeverity,
@ErrorState, @ErrorProcedure, @ErrorLine, @ErrorMessage);

End Catch;

GO
```