Red Gate Books

# SQL Server DMV Starter Pack

Glenn Berry, Louis Davidson and Tim Ford

# SQL Server DMV Starter Pack

By Glenn Berry, Louis Davidson, Tim Ford

First published by Simple Talk Publishing July 2010

ISBN 978-1-906434-51-9

Editor: Tony Davis
Additional Material: Adam Machanic
Typeset and Designed by Matthew Tye
Cover Photograph by Paul Vlaar

# Table of Contents

About the Contributors .......... v
Glenn Berry (Author) .......... v
Louis Davidson (Author) .......... v
Tim Ford (Author) .......... v
Adam Machanic (Additional Material) .......... vi
Introducing Dynamic Management Objects .......... 7
What sort of data is available? .......... 8
Point in time versus cumulative .......... 9
Performance Tuning with DMOs .......... 11
Connections, Sessions, Requests, Queries .......... 15
Optimizing your SQL Workload .......... 16
DMV#1: Are you Connected? .......... 17
DMV#2: Session Ownership .......... 19
DMV#3: Current expensive, or blocked, requests .......... 20
DMV#4: Query Stats – Find the "top X" most expensive cached queries .......... 21
DMV#5: How many single-use ad-hoc Plans? .......... 23
DMV#6: Ad-hoc queries and the plan cache .......... 24
DMV#7: Investigate expensive cached stored procedures .......... 26
DMV#8: Find Queries that are waiting, or have waited, for a Memory Grant .......... 29
Transactions .......... 31
DMV#9: Monitor long-running transactions .......... 32
DMV#10: Identify locking and blocking issues .......... 33
Databases and Indexes .......... 36
DMV#11: Find Missing Indexes .......... 37
DMV#12: Interrogate Index Usage .......... 39
DMV#13: Table Storage Stats (Pages and Row Counts) .......... 43
DMV#14: Monitor TempDB .......... 44

**Disk I/O** ........ **46**
DMV#15: Investigate Disk Bottlenecks via I/O Stalls ........ 47
DMV#16: Investigate Disk Bottlenecks via Pending I/O ........ 48

**Operating System** ........ **50**
A Brief Overview of SQL Server Waits ........ 50
DMV#17: Why are we Waiting? ........ 52
DMV#18: Expose Performance Counters ........ 55
DMV#19: Basic CPU Configuration ........ 56
DMV#20: CPU Utilization History ........ 57
DMV#21: Monitor Schedule activity ........ 59
DMV#22: System-wide Memory Usage ........ 60
DMV#23: Detect Memory Pressure ........ 61
DMV#24: Investigate Memory Usage Across all Caches ........ 63
DMV#25: Investigate memory use in the Buffer Pool ........ 64

**Other Useful DMVs** ........ **67**
DMV#26: Rooting out Unruly CLR Tasks ........ 67
DMV#27: Full Text Search ........ 68
DMV#28: Page Repair attempts in Database Mirroring ........ 70

**Conclusion** ........ **72**

# About the Contributors

## Glenn Berry (Author)

Glenn Berry works as a Database Architect at NewsGator Technologies in Denver, CO. He is a SQL Server MVP, and has a whole collection of Microsoft certifications, including MCITP, MCDBA, MCSE, MCSD, MCAD, and MCTS, which proves that he likes to take tests. He is also an Adjunct Faculty member at University College, University of Denver, where he has been teaching since 2000, and he has completed the Master Teacher Program. His blog is at HTTP://GLENNBERRYSQLPERFORMANCE.SPACES.LIVE.COM and he is GlennAlanBerry on Twitter.

## Louis Davidson (Author)

Louis has been in the IT industry for 16 years as a corporate database developer and architect. Louis has been a SQL Server Microsoft MVP for 6 years and has written 4 books on database design. Currently he is the Data Architect and sometimes DBA for the Christian Broadcasting Network supporting offices in Virginia Beach, Virginia and Nashville, Tennessee. He graduated from the University of Tennessee at Chattanooga with a Bachelor's degree in Computer Science with a minor in mathematics.

For more information, visit his website at drsql.org or email him at LOUIS@DRSQL.ORG.

## Tim Ford (Author)

Timothy is a SQL Server MVP and has been working with SQL Server for over 10 years. He is the Primary DBA and Subject Matter Expert for the SQL Server platform, for Spectrum Health.  He's been writing about technology since 2007 for a variety of websites and maintains his own blog at WWW.THESQLAGENTMAN.COM, covering SQL as well as telecommuting and professional development topics. Tim also dedicates content to his other passions: Photography, Cooking, Fitness, Travel, and Gaming. He recently launched SQL Cruise (www.sqlcruise.com), a training company for SQL Server specializing in deep dive sessions to small groups hosted in exotic and alternative locations throughout the world. Whenever possible, he loves to spend time with his wife, Amy and sons: Austen and Trevor, doing anything other than thinking about relational databases.

## Adam Machanic (Additional Material)

Adam Machanic is a Boston-based independent database consultant, writer, and speaker. He has been involved in dozens of SQL Server implementations for both high-availability OLTP and large-scale data warehouse applications, and has optimized data access layer performance for several data-intensive applications. Adam has written for numerous web sites and magazines, and has contributed to several books on SQL Server, including "SQL Server 2008 Internals" (Microsoft Press, 2009) and "Expert SQL Server 2005 Development" (Apress, 2007). Adam regularly speaks at user groups, community events, and conferences, and is a Microsoft Most Valuable Professional (MVP) for SQL Server and Microsoft Certified IT Professional (MCITP).

Adam is the author of a free monitoring stored procedure – said by some to be the most comprehensive single DMV script ever written – called "Who is Active". The most recent updates are available at http://www.tinyurl.com/whoisactive.

## Introducing Dynamic Management Objects

Dynamic Management Objects (DMOs) are a set of SQL Server objects, stored in the **system** schema, which provide the SQL Server DBA with a window into the activities being performed on their SQL Server instances, and the resources that this activity is consuming. In other words, these DMOs expose valuable information concerning the connections, sessions, transactions, SQL statements and processes that are executing against a database instance, the resultant workload generated on the server, how it is distributed, where the pressure points are, and so on. Having revealed a particular pressure point on their SQL Server, the DBA can then take appropriate steps to alleviate the problem, perhaps by tuning a query, adding an index, re-spec'ing the disk subsystem, or simply 'killing' a blocking session.

The term "dynamic" refers to the fact that the information stored in these DMOs is generated dynamically from a vast range of 'instrumentation' points, in memory structures throughout the SQL Server engine. This data is then exposed in tabular form in the **sys** database schema, either in views, in which case they are referred to as **Dynamic Management Views** (DMVs), or in table-values functions, in which case they are referred to as **Dynamic Management Functions** (DMFs).

***Too many Acronyms***

*The objects, collectively, should be referred to as DMOs, but since it tends to cause some confusion with the entirely-unrelated "Distributed Management Objects", it's still very common for DBAs to refer to the Dynamic Management Objects, collectively, as "DMVs".*

So, DMVs and DMFs are simply system views and system functions, and you use them just like you use any other view and function within SQL Server: querying them, joining, passing parameters, and ultimately returning a single result set containing the data you need to investigate a particular issue regarding the state or health of your SQL Server instance.

The goal of this short book is to provide a brief overview of how and where the DMOs can be used, where they fit alongside the number of other performance tools available to the DBA, and then to provide a "starter pack" of useful scripts with which to diagnose common SQL Server performance issues.

***Performance Tuning with SQL Server DMVs***

*Note that this "starter pack", as the name implies, is in no way intended to a comprehensive to guide to the information that these DMOs can provide; indeed, we only scratch the surface of the sort of information that is available. For more in-depth coverage of the DMOs, please check out the* FORTHCOMING BOOK BY LOUIS DAVIDSON AND TIM FORD

## What sort of data is available?

The DMOs expose a sometimes-dizzying array of information; the original **sysprocesses** system view has essentially been de-normalized, and many new DMOs have been added, and many new data columns been made available for querying. As the database engine becomes better-and-better instrumented, so the amount of data available about the engine, and the work it is doing, will continue to grow.

The added complexity of stitching together data from a disparate array of DMOs, coupled with the initially-baffling choices of what columns will be exposed where, has lead some DBAs to liken querying DMOs to "collecting mystic spells". However, the "de-normalization" process has, in many ways, made the data that the DMOs return much easier to analyze and understand, and once you start to write your own scripts, you'll see the same tricks, and similar join patterns, being used time and again. As such, a relatively small, core set of scripts can be readily adapted to suit many requirements.

In some ways, working through the DMOs for the diagnostic data you need is a process of "peeling back layers". At the outer layer, we can find out who is **connected** to our SQL Server instances, and how; what **sessions** are running against them, and what **requests** are being performed by these sessions. From here, we can find out the details of the **SQL statements** being executed by these requests, the **query plans** that are being used to run them, and so on. Dropping down a layer, we have the **transaction** level, where we can find out what **locks** are being held as a result of these transactions, investigate any potential **blocking**, and so on. Moving down another layer, we can find how the workload represented by the submitted requests translates into actual work in the operating system. We can find out, for example:

- What actual tasks (**threads**) are being executed in order to fulfill the requests
- What work they are performing in terms of I/O, CPU and memory usage
- How I/O is distributed among the various files
- How long threads spend waiting, unable to proceed, and why

It is the job of the DBA to join together all the pieces of data, from the various different layers, to provide the results needed to highlight the specific problems in the system.

## Point in time versus cumulative

As noted, we can query data held on the DMOs just as we would any other table, view or function. However, always remember that the data you are seeing is “dynamic” in nature. It is collected from a range of different structures in the database engine and represents, in the main, a point-in-time “snapshot” of the activity that was occurring on your server at the time you run the DMO query.

Sometimes, this is exactly what you want; you have a performance issue, and want to find out what queries are running right now on the server that could be causing it. Sometimes though, you may find it quite difficult to query the data in these point-in-time DMOs in the hope that the problem will simply "jump out at you". If, for example, you have a performance problem and what to check for any "unusual" locking patterns, then it's unlikely that a "*select [columns] from [locking DMV]*" will tell you much, unless you're very familiar with what "normal" locking looks like on your system, and you can easily spot anomalies.

Bear in mind, also, that the point-in-time data can and likely will change each time you query it, as the state of the server changes. You should expect to occasionally see anomalous or non-representative results and you may need to run a script many times to get a true picture of activity on your instance.

In other cases, the DMOs are cumulative. In other words, the data in a given column is accumulative and incremented every time a certain event occurs. For example, every time a session waits a period of time for a resource to become available, this is recorded in a column of the `sys.dm_os_wait_stats` DMV. When querying such a DMV, you will be seeing, for example, the total amount of time spent waiting for various resources, across all sessions, since SQL Server was started or restarted – unless a DBCC command was run to manually clear out the stored statistics. While this will give you a broad overview of where time has been spent waiting, over a long period, it will make it hard to see the smaller details. If you want to measure the impact of a certain change to the database (a new index for example), you'll need to take a baseline measurement, make the change, and then measure the difference.

Finally, always bear in mind that much of the data you're seeing in such DMOs is aggregate data, collected across many sessions, many requests and many transactions. The previously mentioned `wait_stats` DMV, for example, will show you at an instance level where SQL Server spent time waiting, aggregated across all sessions. You cannot track the wait times at an individual session level (unless you're working on an isolated server!).

## Performance Tuning with DMOs

In simplistic terms, performance problems are caused by excessive demands on some shared resource of the SQL Server system, leading to bottlenecks and poor response times. The biggest problem for most DBAs is in pinpointing the exact cause of the problem. There are many "shared resources" in SQL Server, from CPU to memory (buffer cache, plan cache etc), to the disk subsystem, to process schedulers and so on. Furthermore, many times a single piece of performance data, considered in isolation, can lead the unwary DBA to misdiagnose a performance problem.

Too often slow query performance is diagnosed as a need for more CPU, or faster disks, without truly knowing the exact cause of the slow performance. If, on your SQL Server instance, 90% of the total response time consists of I/O waits and only 10% of the time is spent on the CPU, then adding CPU capacity or upgrading to faster CPUs won't have the desired impact. The DMOs allow you to narrow the focus of tuning and troubleshooting quickly, and then to accurately diagnose the true problem with further investigation with DMOs, and other available performance tools.

**Profiler**, for example, is an invaluable tool for tracing a defined set of actions (events) that are occurring in SQL Server in response to a certain SQL workload. It is a powerful tool for diagnosing poorly performing queries, locking, blocking, and a lot more. Indeed, DBAs will continue to use Profiler regularly, alongside the DMOs. The DMOs tell you a great deal about the state of your instance and provide, as you'll see early in the book, significant information about the sessions, requests, and transactions that comprise the activity in your SQL Server instance. However, Profiler is still unequalled in its ability to offer real-time insight into the activity on your instance from SQL Profiler. It is not a lightweight tool though, and you need to have the most-concise set of filters and return the correct columns and rows for your trace in order to prevent the "watcher" effect, whereby collecting the performance data actually has a detrimental impact on performance. In this respect, the DMOs are often useful in providing initial, detailed, rapid insight into where we should focus the scalpel on the injured athlete that is our SQL Server instance.

DMOs do offer further advantages over Profiler. One of the limitations of Profiler is that it can only collect performance statistics while it is running, actively capturing query activity. If a "bad query" suddenly executes on the server, causing performance issues, the DBA will usually find out about it after the event. He or she will need to run a trace and hope the problem occurs again so that it can be captured and analyzed. With the DMOs, as long as the query plan is still in the cache, you can retrieve it and find out what resources the queries used when it ran, and who ran it. Another advantage is the fact that running DMO queries will, in general, have a much lower impact on the server than running Profiler traces. Finally, for certain issues, DMOs offer a much more granular level of detail than Profiler. For example, with DMOs, we can investigate IO activity at the file system level, whereas Profiler will only summarize IO activity at the drive level.

**Performance Monitor** (PerfMon) allows us to graphically identify a large number of metrics, but primarily at the server level. Remember that PerfMon is first-and-foremost a Windows troubleshooting tool. There are Microsoft SQL Server Performance objects that you can utilize to look at SQL-centric metrics but, again, they are not at the same level of granularity that you get from the DMOs.

Wait statistics are a prime example. Each time a worker needs to wait for a resource, it is recorded in SQL Server. This information is cached and incremented until the next SQL Server service restart. We can query and aggregate these metrics via `sys.dm_os_wait_stats`:

```
SELECT  wait_type ,
        SUM(wait_time_ms / 1000) AS [wait_time_s]
FROM    sys.dm_os_wait_stats DOWS
WHERE   wait_type NOT IN ( 'SLEEP_TASK', 'BROKER_TASK_STOP',
                           'SQLTRACE_BUFFER_FLUSH', 'CLR_AUTO_EVENT',
                           'CLR_MANUAL_EVENT', 'LAZYWRITER_SLEEP' )
GROUP BY wait_type
ORDER BY SUM(wait_time_ms) DESC
```

As will be discussed later in the book, aggregating and sorting upon wait_type and wait_time_ms will allow us to prioritize our troubleshooting, by identifying what type of waits the SQL Server instance is encountering. From there we can use other DMOs to further narrow our tuning scope. At this point, we may also choose to incorporate data extracted from Profiler, PerfMon, or our chosen third-party performance monitoring tool.

For example, if the top waits consisted of CXPACKET and SOS_SCHEDULER_YIELD, we'd need to look at CPU concerns, specifically parallelism settings. By default, SQL Server allows all processors to partake in parallel queries. However, it is unusual for OLTP queries to benefit much from parallel execution. We recently saw behavior identical to this on one of the nodes that host 50 of our databases, that had been consolidated onto a two node cluster. On changing the instance's Max Degree of Parallelism setting from 0 (dynamic) to 4 (this was an eight core server), the CXPACKET waits associated with parallelism drop off the top of the list. If we were to attempt the same analysis with PerfMon, we would have had to play that game DBAs used to always have to play in the past: pick a Counter. The SQLServer:Wait Statistics Performance Object has a dozen counters associated with it. If someone were to ask me why I recommend using the DMVs to ascertain performance issues in my environment, and why they are the first tool I grab in my toolbox when I don't know why an instance in performing badly, I point to this as my prime example.

Depending upon the results of queries against sys.dm_os_wait_stats, you will probably investigate the particular issue indicated, be it CPU, Memory, I/O or other specific concerns, further with the DMOs, correlate the data with that obtained from PerfMon, run profiler traces, use third party monitoring tools, and so on, until the exact cause is diagnosed.

**Activity Monitor** is one native tool, built into SSMS, which uses DMOs (as well as PerfMon counters) under the covers. Graphical representations of % CPU Time, Waiting Tasks Count, Database I/O MB/sec, and Batch Requests/sec – counters you would typically enlist first in PerfMon – are included along with output from queries against sys.dm_exec_sessions, sys.dm_exec_requests, sys.dm_os_wait_stats, sys.dm_exec_query_stats, and many others. These are segregated into categories for Processes, Waits, Data File I/O, and Recent Query Costs. Activity Monitor is not a bad place to start a performance investigation, but just doesn't offer the same degree of control and filtering that you'll get from using DMOs and good old T-SQL.

Some DBAs still use tools such as **sp_who2**, or some of the **DBCC** commands, to extract similar information to that which you can get from a few of the DMOs. Again, these former tools are simply rather limited in the information they return, compared to the DMOs. For example, `DBCC OPENTRAN` can return some of the information that you can get from the transaction-related DMOs, with regard to active transactions in the system. However, the DBCC command fails to give a complete picture, missing for example 'sleeping' transactions that may still be holding locks, and also not providing valuable information such as how many log records or how many bytes have been written by a given transaction, which is very useful when diagnosing a rapidly-filling transaction log file.

Finally, many **third-party monitoring tools** have dashboards that will tie directly into the DMOs, along with functionality to allow for tracing activity because these two tools are so vital for rapid, successful identification and remediation of performance issues in Microsoft SQL Server. Such tools can remove some of the pain of having to construct and maintain custom scripts, schedule them, and collate the results in meaningful form, and so on. Such tools will scrape the data out of the DMOs "under the covers", present it to the DBA in a digestible form and, critically, warn in advance of impending problems. Of course, the DBA will still want to know how things work under the covers, will still need to occasionally write their own custom queries directly against the DMVs, because what you lose with a tool, inevitably, is the exact granularity and control of the data returned, that you get when writing your own queries.

Ultimately, it is the granularity of the data, and level of programmatic control, which makes the DMOs such an important addition to the DBA's performance tuning toolkit. DMOs do not necessarily replace other performance tools, but they do offer a level of detail that is largely unavailable elsewhere, or at least from the native tool set.

# Connections, Sessions, Requests, Queries

All of the DMOs covered in this section belong to the "Execution related" category of Dynamic Management objects, and consequently the name of each view in this section begins with "`sys.dm_exec_`". The DMOs in this category can be considered as comprising the "top level" of our performance tuning investigations, and essentially we are interested in the following information (example DMOs that can provide it are listed in brackets):

- The connections that are accessing our database instances and who/what owns them (**`connections`**)
- The sessions that are spawned inside these connections (**`sessions`**)
- The requests that are executed by these sessions, for work to be performed by SQL Server. These requests can comprise a single query, a batch, a call to a stored procedure and so on. (**`requests`**)
- The actual SQL that is executed as a result of running these requests (**`sql_text`**)
- The execution plans used by the Optimizer in order to run the queries and stored procedures (**`query_plan`**, **`cached_plans`**)
- The time these queries take to execute, and the work that these queries and stored procedures, in terms of I/O, CPU, memory usage (**`query_stats`**, **`procedure_stats`**, **`query_memory_grants`**)

In this chapter, we'll provide some useful scripts that you can use to extract this information. Unless stated otherwise, all of the scripts in this section work with SQL Server 2005, 2008, and 2008 R2 and all require `VIEW SERVER STATE` permission.

## Optimizing your SQL Workload

Ultimately, query tuning is the heart and soul of optimizing SQL Server performance. If your typical workload consists of ill-designed, inefficient queries then you will have performance and scalability issues, for a number of reasons.

If your queries are longer, more numerous, and more complex than necessary, they will require more CPU resources during execution, and so will take longer to run. Ill-designed queries, along with a failure to make proper use of indexes, will lead to more data being read more often than is necessary. If this data is read from the buffer cache, this is referred to as logical I/O, and can be an expensive operation. If the data is not in memory, and so needs to be read from disk (or, of course, if data needs to be written), this is physical I/O and is even more expensive. In addition, if you have many queries that return huge amounts of data, it could cause memory pressure on the buffer cache, and result in SQL Server flushing data out of the cache, which will affect the performance of other queries. A "golden rule" of well-designed SQL is to return no more data than you really need, to pass through the data as few times as possible and to use set-based logic to manipulate that data into the result set you need.

Parsing and optimizing SQL Statements is not a "high concurrency" operation. SQL Server stores plans for previously-executed queries in a shared memory area called the plan cache. Whenever a query is submitted for execution, SQL Server checks in the plan cache to see if it can use an existing plan to execute the query. Every time it cannot find a match, the submitted query must be parsed, optimized, and a plan generated. This is a CPU-intensive process. Furthermore, each time it does this, SQL Server acquires latches on the plan cache to protect the relevant area of memory from other updates. More ad-hoc, non-parameterized SQL means more single-use plans in the cache, more CPU consumed and latches acquired during parsing, and ultimately a non-scalable system. Well-designed SQL will promote plan reuse ("parse once, use many times" as far as possible.

Ultimately, if your workload consists of poorly-designed queries, then they will cause needless extra I/O, CPU, and memory overhead, and execution times will be slow. The situation will get worse and worse as the number of users grows, and their requests are forced to wait for access to the shared resources that your queries are monopolizing. Conversely, if you can minimize the number of individual SQL statements you need to get a particular job done, and then minimize the work done by each of those individual SQL statements, you are much more likely to have a fast, responsive SQL Server system, which scales gracefully as the number of users grows. The set of DMVs in the "Execution-related" category can help you achieve this goal, by allowing you to track down the sessions, requests and queries that are the most resource-intensive, and take the longest time to execute.

As such, an often-used approach to performance tuning, using these DMVs, is to retrieve a "top X" list of the slowest queries that constitute part of the normal, daily workload on your SQL Server instance and then tune them, one by one. Arguably, a slightly more scientific approach might start at the lower levels (see the later sections on OS and I/O), looking for specific areas where SQL Server is experiencing resource pressure, where processes are waiting unusually long times for some other action to complete before proceeding. In this way, you can work out whether the major component of the slow execution time is CPU time (i.e. the system is CPU-bound), or time spent waiting for I/O (I/O-bound), and so on. The DBA can then work back from there to the requests that are causing the resource contention. Either way, having isolated the problem queries, the DBA must then find a way to reduce the amount of work being performed, usually by tuning the SQL, or by adding indexes, or if all else fails, by buying more memory/disk/CPU power.

## DMV#1: Are you Connected?

The **`sys.dm_exec_connections`** DMV is described by Books Online (BOL) as follows:

> Returns information about the connections established to this instance of SQL Server and the details of each connection.

The most immediately-obvious use for this DMV is to help identify who and what is connecting to an instance of SQL Server, and to gather some useful information about each connection.

```sql
-- Get a count of SQL connections by IP address
SELECT  ec.client_net_address ,
        es.[program_name] ,
        es.[host_name] ,
        es.login_name ,
        COUNT(ec.session_id) AS [connection count]
FROM    sys.dm_exec_sessions AS es
        INNER JOIN sys.dm_exec_connections AS ec
                                   ON es.session_id = ec.session_id
GROUP BY ec.client_net_address ,
        es.[program_name] ,
        es.[host_name] ,
        es.login_name
ORDER BY ec.client_net_address ,
        es.[program_name] ;
```

Script 1: Querying `sys.dm_exec_connections` to find out who is connected

This particular query provides the IP address and name of the machine form which the connection is being made, to a given SQL Server instance, the name of the program that is connecting and the number of open sessions for each connection.

I find this to be extremely useful information in several ways. It lets you see if anyone is using SQL Server Management Studio (SSMS) to connect to your instance. It lets you see which middle-tier servers are connecting to your server, and how many sessions each one of them has, which is very helpful when you are trying to help your developers debug application or connectivity issues.

## DMV#2: Session Ownership

This script uses the **`sys.dm_exec_sessions`** DMV, which is described by BOL as follows:

"Returns one row per authenticated session on SQL Server. *`sys.dm_exec_sessions`* is a server-scope view that shows information about all active user connections and internal tasks. This information includes client version, client program name, client login time, login user, current session setting, and more."

The simple query, DMV2, reports on the number of sessions being run by each login on your SQL Server instance.

```
--  Get SQL users that are connected and how many sessions they have
SELECT  login_name ,
        COUNT(session_id) AS [session_count]
FROM    sys.dm_exec_sessions
GROUP BY login_name
ORDER BY COUNT(session_id) DESC ;
```

Script 2: Which logins a running which sessions?

This can be useful, especially if you use application level logins for different applications that use your database instance. If you know your baseline values for the number of connections per login, it is easier to see when something has changed.

An interesting point to note is that the values in the `sessions` DMV are updated only when their associated sessions have finished executing, whereas the `requests` DMV (covered next) provides a "real time" view of what is happening right now on your system. If a request is finished, it will cease to appear in the requests DMV, even if the transaction with which it is associated is still uncommitted.

## DMV#3: Current expensive, or blocked, requests

This script makes use of two DMOs. The first is the **sys.dm_exec_requests** DMV, which is described by BOL as follows:

"Returns information about each request that is executing within SQL Server"

The seconds is the **sys.dm_exec_sql_text** DMF, which is described by BOL as follows:

"Returns the text of the SQL batch that is identified by the specified *sql_handle*. This table-valued function replaces the system function *fn_get_sql*."

Together, we can use these DMVs to get a quick snapshot of currently executing requests on a given instance of SQL Server, to find out any that are particularly expensive, or that are blocked for some reason.

```
-- Look at currently executing requests, status and wait type
SELECT  r.session_id ,
        r.[status] ,
        r.wait_type ,
        r.scheduler_id ,
        SUBSTRING(qt.[text], r.statement_start_offset / 2,
            ( CASE WHEN r.statement_end_offset = -1
                   THEN LEN(CONVERT(NVARCHAR(MAX), qt.[text])) * 2
                   ELSE r.statement_end_offset
              END - r.statement_start_offset ) / 2) AS [statement_
executing] ,
        DB_NAME(qt.[dbid]) AS [DatabaseName] ,
        OBJECT_NAME(qt.objectid) AS [ObjectName] ,
        r.cpu_time ,
        r.total_elapsed_time ,
        r.reads ,
        r.writes ,
        r.logical_reads ,
        r.plan_handle
```

```
FROM    sys.dm_exec_requests AS r
        CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS qt
WHERE   r.session_id > 50
ORDER BY r.scheduler_id ,



        r.[status] ,
        r.session_id ;
```

Script 3: Current expensive, or blocked, requests

This script uses a couple of query patterns that are very common when using the DMVs. The first is the retrieval of a binary handle, called `sql_handle`, from one DMV, in this case `sys.dm_exec_requests`, and passing it via a `CROSS APPLY`, to the `sys.dm_exec_sql_text` DMF to extract the SQL text of the executing batch. This, in itself, leads to the second common pattern: namely the use of the `SUBSTRING` function and byte offset values (`statement_start_offset` and `statement_end_offset`) to extract from the batch, which may consist of tens or even hundreds of SQL statements, the text of the statement within that batch that is currently-executing. The division by two is to handle the Unicode to string conversion. Very similar patterns will occur when retrieving execution plans using the `plan_handle`, and then extracting the plan for the current query only.

I like to periodically run this query multiple times against an instance to get a "feel" for what queries and stored procedures are regularly encountering which types of waits, and which ones are expensive in different ways. Unless you have a particularly long running query, the output will be different each time you run this query on a busy server.

## DMV#4: Query Stats – Find the "top X" most expensive cached queries

This script uses the **`sys.dm_exec_query_stats`** DMV, which is described in BOL as follows:

"Returns aggregate performance statistics for cached query plans. The view contains one row per query statement within the cached plan, and the lifetime of the rows are tied to the plan itself. When a plan is removed from the cache, the corresponding rows are eliminated from this view. An initial query of *sys.dm_exec_query_stats* might produce inaccurate results if there is a workload currently executing on the server. More accurate results may be determined by rerunning the query."

Script 4 assumes that the DBA has identified his system as "CPU bound" and uses the `total_worker_time` column to find out which queries the server is spending the most time executing.

```
SELECT TOP (3)
        total_worker_time ,
        execution_count ,
        total_worker_time / execution_count AS [Avg CPU Time] ,
        CASE WHEN deqs.statement_start_offset = 0
                  AND deqs.statement_end_offset = -1
             THEN '-- see objectText column--'
             ELSE '-- query --' + CHAR(13) + CHAR(10)
                  + SUBSTRING(execText.text, deqs.statement_start_offset / 
2,
                              ( ( CASE WHEN deqs.statement_end_offset = -1
                                       THEN DATALENGTH(execText.text)
                                       ELSE deqs.statement_end_offset
                                  END ) - deqs.statement_start_offset ) / 
)
        END AS queryText
FROM    sys.dm_exec_query_stats AS deqs
        CROSS APPLY sys.dm_exec_sql_text(deqs.plan_handle) AS execText
ORDER BY deqs.total_worker_time DESC ;
```

Script 4: Top 3 CPU-sapping queries for which plans exist in the cache

Note that it isn't enough just to know that the server is spending a lot of time executing a particular query. In fact, without context this piece of information is more or less meaningless. It might be that the query is run a million times, and no other query is executed more than a thousand times. So, to add the required context, we included the `execution_count`, along with a calculation of the average CPU time.

One could write a similar query to investigate an I/O-bound system, using (`total_logical_reads` + `total_logical_writes`)/`execution_count` in the calculation. This would identify the most I/O-intensive queries, on which one could focus tuning efforts and indexing strategy.

## DMV#5: How many single-use ad-hoc Plans?

This script uses the `sys.dm_exec_cached_plans` DMV which is described by BOL as follows:

> "Returns a row for each query plan that is cached by SQL Server for faster query execution. You can use this dynamic management view to find cached query plans, cached query text, the amount of memory taken by cached plans, and the reuse count of the cached plans."

Script 5 provides the use counts for each compiled plan in the plan cache, broken out by the type of plan.

```
-- Use Counts and # of plans for compiled plans
SELECT  objtype ,
        usecounts ,
        COUNT(*) AS [no_of_plans]
FROM    sys.dm_exec_cached_plans
WHERE   cacheobjtype = 'Compiled Plan'
GROUP BY objtype ,
        usecounts
ORDER BY objtype ,
        usecounts ;
```

Script 5: Number of cached plans, and their use counts

Pay close attention to ad-hoc plans, where the use count is 1. Looking at the numbers and use counts of your other types of cached plans (such as proc and prepared) can help you better understand your workload. You may be surprised to find out that you get more ad-hoc queries than you expect.

## DMV#6: Ad-hoc queries and the plan cache

This script uses two DMOs previously described, namely **`sys.dm_exec_cached_plans`** and **`sys.dm_exec_sql_text`**. Having gained a broad overview of your single-use plans, and their type, in DMV#5, Script 6 allows you to retrieve the text for each single-use plan that is bloating your plan cache.

```
-- Find single-use, ad-hoc queries that are bloating the plan cache
SELECT TOP ( 100 )
        [text] ,
        cp.size_in_bytes
FROM    sys.dm_exec_cached_plans AS cp
        CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE   cp.cacheobjtype = 'Compiled Plan'
        AND cp.objtype = 'Adhoc'
        AND cp.usecounts = 1
ORDER BY cp.size_in_bytes DESC ;
```

Script 6: Seeking out single-use plans in the cache

This query will identify ad-hoc queries that have a use count of 1, ordered by the size of the plan. It gives you the text and size of single-use ad-hoc queries that waste space in plan cache. This usually happens when your developers build T-SQL commands by concatenating a variable at the end of a "boilerplate" T-SQL statement. A very simplified example is shown below:

```
-- Query 1
SELECT  FirstName ,
        LastName
FROM    dbo.Employee
WHERE   EmpID = 5

-- Query 2
SELECT  FirstName ,
        LastName
FROM    dbo.Employee
WHERE   EmpID = 187
```

Even though these two queries are essentially identical, they might each have a separate plan in the cache, just because the literal value is different in each case. Actually these two queries are so simple (with no joins) that SQL Server would probably parameterize them, even using just the default SIMPLE PARAMETERIZATION (as opposed to FORCED PARAMETERIZATION). Following is a more realistic example of how not to write SQL:

```
  DECLARE @SortColumn DATETIME
  DECLARE @LastValue INT
  SET ROWCOUNT 1
  SELECT    @SortColumn = ISNULL(InstantForum_Topics.DateStamp, '') ,
            @LastValue = InstantForum_Topics.PostID
  FROM      InstantASP_Users (NOLOCK)
            INNER JOIN InstantForum_Users
                  ON InstantASP_Users.UserID = InstantForum_Users.UserID
            RIGHT OUTER JOIN InstantForum_Messages
            INNER JOIN InstantForum_Topics
                  ON InstantForum_Messages.PostID = InstantForum_Topics.
PostID
                  ON InstantForum_Users.UserID = InstantForum_Topics.UserID
  WHERE     InstantForum_Topics.TopicID = 34568
            AND InstantForum_Topics.Queued <> 1
  ORDER BY  InstantForum_Topics.DateStamp ,
            InstantForum_Topics.PostID
  SET ROWCOUNT 21
```

This is a very poorly-written ad-hoc SQL query that is unlikely to be called again very often. Nevertheless, it ends up being in the plan cache with a use count of 1, wasting space.

Enabling 'optimize for ad hoc workloads' for the instance can help (SQL Server 2008 only). Enabling forced parameterization for the database can also help, but you should test that in your environment.

## DMV#7: Investigate expensive cached stored procedures

The next script uses the **`sys.dm_exec_procedure_stats`** DMV, which is described by BOL as follows:

> "Returns aggregate performance statistics for cached stored procedures. The view contains one row per stored procedure, and the lifetime of the row is as long as the stored procedure remains cached. When a stored procedure is removed from the cache, the corresponding row is eliminated from this view. At that time, a Performance Statistics SQL trace event is raised similar to *`sys.dm_exec_query_stats`*."

This DMV is new to SQL Server 2008 so the following scripts will only work on SQL Server 2008 and 2008 R2. You can get similar data out of `sys.dm_exec_cached_plans`, which will work on SQL Server 2005, just not quite as easily.

This DMV allows you to discover a lot of very interesting and important performance information about your cached stored procedures.

```
-- Top Cached SPs By Total Logical Reads (SQL 2008 only).
-- Logical reads relate to memory pressure
SELECT TOP ( 25 )
        p.name AS [SP Name] ,
        qs.total_logical_reads AS [TotalLogicalReads] ,
        qs.total_logical_reads / qs.execution_count AS [AvgLogicalReads] ,
        qs.execution_count ,
        ISNULL(qs.execution_count /
                 DATEDIFF(Second, qs.cached_time, GETDATE()),
                0) AS [Calls/Second] ,
        qs.total_elapsed_time ,
        qs.total_elapsed_time / qs.execution_count AS [avg_elapsed_time] ,
        qs.cached_time
FROM    sys.procedures AS p
        INNER JOIN sys.dm_exec_procedure_stats AS qs
                      ON p.[object_id] = qs.[object_id]
WHERE   qs.database_id = DB_ID()
ORDER BY qs.total_logical_reads DESC ;
```

Script 7: Investigating logical reads performed by cached stored procedures

Depending on what columns you include, and which column you order by, you can discover which cached stored procedures are the most expensive from several different perspectives. In this case, we are interested in finding out which stored procedures are generating the most total logical reads (which relates to memory pressure). I always like to run this query, but I would especially want to run it if I saw signs of **memory pressure**, such as a persistently low page life expectancy and/or persistent values above zero for memory grants pending. This query is filtered by the current database, but you can change it to be instance wide by removing the `WHERE` clause.

Simply by selecting the `total_phyisical_reads` column, instead of `total_logical_reads`, in this query, we can perform the same analysis from the perspective of physical reads, which relates to read, disk I/O pressure.

```
-- Top Cached SPs By Total Physical Reads (SQL 2008 only)
-- Physical reads relate to disk I/O pressure
SELECT TOP ( 25 )
        p.name AS [SP Name] ,
        qs.total_physical_reads AS [TotalPhysicalReads] ,
        qs.total_physical_reads / qs.execution_count AS [AvgPhysicalReads]
,
        qs.execution_count ,
        ISNULL(qs.execution_count /
                 DATEDIFF(Second, qs.cached_time, GETDATE()),
               0) AS [Calls/Second] ,
        qs.total_elapsed_time ,
        qs.total_elapsed_time / qs.execution_count AS [avg_elapsed_time] ,
        qs.cached_time
FROM    sys.procedures AS p
        INNER JOIN sys.dm_exec_procedure_stats AS qs
                      ON p.[object_id] = qs.[object_id]
WHERE   qs.database_id = DB_ID()
ORDER BY qs.total_physical_reads DESC ;
```

Script 8: Investigating physical reads performed by cached stored procedures

If you see lots of stored procedures with high total physical reads or high average physical reads, it could actually mean that you are under severe memory pressure, causing SQL Server to go to the disk I/O subsystem for data. It could also mean that you have lots of missing indexes or that you have "bad" queries (with no `WHERE` clauses for example) that are causing lots of clustered index or table scans on large tables.

Be aware though, that there are a couple of caveats with these queries. The big one is that you need to pay close attention to the `qs.cached_time` column as you compare rows in the result set. If you have stored procedures that have been cached for different periods of time, then this will skew the results.

One easy, but perhaps controversial, solution to this problem is to periodically clear your procedure cache, by running `DBCC FREEPROCCACHE` with a SQL Agent job. That will cause all of your stored procedures to recompile and so flushes every plan out of the cache for the entire instance. These plans will go back into the cache the next time they are executed, the result being that most of the stored procedures that are run frequently and are part of your normal workload will have a similar cache time. This will make these queries much easier to interpret (until and unless they get recompiled again for some other reason).

I say "perhaps controversial" because you've probably seen many warnings to avoid running `DBCC FREEPROCCACHE`, or even its more selective cousin `DBCC FREEPROCINDB(db_id)`, on a production server. It's true that recompiling all of the query plans will cause some extra work for your processor(s), but in my experience, modern processors shrug this off with almost no measurable effect beyond a very brief (a few seconds) spike of CPU activity. Overall, I think the benefits you gain, which will include flushing the single-use ad-hoc query plans (see DMV#5) out of the cache, far outweigh the very slight amount of extra CPU required to recompile the query plans.

The second caveat is that only cached stored procedures will show up in these queries. If you are using `WITH RECOMPILE` or `OPTION(RECOMPILE)`, which is usually not a good idea anyway, then those plans won't be cached.

## DMV#8: Find Queries that are waiting, or have waited, for a Memory Grant

The next script uses the **`sys.dm_exec_query_memory_grants`** DMV, which is described by BOL as follows:

> "Returns information about the queries that have acquired a memory grant or that still require a memory grant to execute. Queries that do not have to wait on a memory grant will not appear in this view."

This DMV allows you to check for queries that are waiting (or recently had to wait) for a memory grant, as demonstrated in script 9. Although not used in this query, note that SQL Server 2008 added some new columns to this DMV.

```
-- Shows the memory required by both running (non-null grant_time)
-- and waiting queries (null grant_time)
-- SQL Server 2008 version
SELECT  DB_NAME(st.dbid) AS [DatabaseName] ,
        mg.requested_memory_kb ,
        mg.ideal_memory_kb ,
        mg.request_time ,
        mg.grant_time ,
        mg.query_cost ,
        mg.dop ,
        st.[text]
FROM    sys.dm_exec_query_memory_grants AS mg
        CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
WHERE   mg.request_time < COALESCE(grant_time, '99991231')
ORDER BY mg.requested_memory_kb DESC ;

-- Shows the memory required by both running (non-null grant_time)
-- and waiting queries (null grant_time)


-- SQL Server 2005 version
SELECT  DB_NAME(st.dbid) AS [DatabaseName] ,
        mg.requested_memory_kb ,
        mg.request_time ,
        mg.grant_time ,
        mg.query_cost ,
        mg.dop ,
        st.[text]
FROM    sys.dm_exec_query_memory_grants AS mg
        CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
WHERE   mg.request_time < COALESCE(grant_time, '99991231')
ORDER BY mg.requested_memory_kb DESC ;
```

Script 9: Which queries have requested, or had to wait for, large memory grants?

You should periodically run this query multiple times in succession, and, ideally, you would want to see few, if any, rows returned each time. If you do see a lot of rows returned each time, then this could be an indication of internal memory pressure.

This query would also help you to identify queries that are requesting relatively large memory grants, perhaps because they are poorly written or they're missing indexes that make the query more expensive.

# Transactions

At the next level down in terms of the "granularity" of our investigation, we have the transaction-related DMOs. All of the DMVs in the "transaction related" category of Dynamic Management Objects begin with "`sys.dm_tran_`". Ultimately, every statement executed against SQL Server is transactional. If you issue a single SQL statement, an implicit transaction is started, under the covers, which starts and auto-completes. If you use explicit `BEGIN TRAN` / `COMMIT TRAN` commands, then you can group together in an explicit transaction a set of statements that must fail or succeed together.

SQL Server implements various transaction isolation levels, to ensure the ACID properties of these transactions. In practical terms, this means that it uses locks and latches to mediate transactional access to shared database resources and prevent 'interference' between the transactions.

Generally speaking, our investigative work in this area is limited to a few key questions:

- What transactions are active, and to what sessions are running them? (**`session_transactions`**, **`active_transactions`**)
- What transactions are doing the most work? (**`database_transactions`**)
- Which transactions are causing locking/blocking problems (**`locks`**)

Of the reasons listed above, investigating locking and blocking is by far the most common use of these DMVs. An area of investigation that will continue to become increasingly common is into the activity generated when using the `SNAPSHOT` isolation level. The snapshot isolation level was introduced in SQL Server 2005. Snapshot isolation eliminates blocking and deadlocking by using a version store in the `tempdb` database to maintain concurrency, rather than establishing locks on database objects. A number of DMVs are provided to investigate this isolation level, but are not covered here.

Let's now move on to the scripts. Again, unless stated otherwise, all of the queries in this section work with SQL Server 2005, 2008, and 2008 R2 and all require `VIEW SERVER STATE` permission.

## DMV#9: Monitor long-running transactions

This script uses two DMVs. The first is **`sys.dm_tran_database_transactions`**, which is described in BOL as follows:

"Returns information about transactions at the database level."

The second is **`sys.dm_tran_session_transactions`**, which simply:

"Returns correlation information for associated transactions and sessions."

The terse description given for `database_transactions` rather belies its potential usefulness. Script 10, for example, provides a query that shows, per session, which databases are in use by a transaction open by that session, whether the transaction has upgraded to read-write in any of the databases (by default most transactions are read-only), when the transaction upgraded to read-write for that database, how many log records written, and how many bytes were used on behalf of those log records.

```
SELECT  st.session_id ,
        DB_NAME(dt.database_id) AS database_name ,
        CASE WHEN dt.database_transaction_begin_time IS NULL THEN 'read-
only'
             ELSE 'read-write'
        END AS transaction_state ,
        dt.database_transaction_begin_time AS read_write_start_time ,
        dt.database_transaction_log_record_count ,
        dt.database_transaction_log_bytes_used
FROM    sys.dm_tran_session_transactions AS st
        INNER JOIN sys.dm_tran_database_transactions AS dt
```

```
            ON st.transaction_id = dt.transaction_id
ORDER BY st.session_id ,
        database_name
```

Script 10: Monitoring transaction activity

These sorts of queries against `database_transactions` are very useful when monitoring, for example:

- Sessions that have open read-write transactions (especially important for sleeping sessions)
- Sessions that are causing the transaction log to grow/bloat
- The progress of long-running transactions (for non-bulk logged operations each effected index row will produce approximately one transaction log record)

## DMV#10: Identify locking and blocking issues

Our example script for the transaction-related category of DMVs uses the **`sys.dm_tran_locks`** DMV, which is described by BOL as follows:

> "Returns information about currently active lock manager resources. Each row represents a currently active request to the lock manager for a lock that has been granted or is waiting to be granted. The columns in the result set are divided into two main groups: resource and request. The resource group describes the resource on which the lock request is being made, and the request group describes the lock request."

This DMV is very useful in helping to identify locking and blocking issues on your database instances.

```
-- Look at active Lock Manager resources for current database
SELECT  request_session_id ,
        DB_NAME(resource_database_id) AS [Database] ,
        resource_type ,
        resource_subtype ,
        request_type ,
        request_mode ,
        resource_description ,
        request_mode ,
        request_owner_type
FROM    sys.dm_tran_locks
WHERE   request_session_id > 50
        AND resource_database_id = DB_ID()
        AND request_session_id <> @@SPID
ORDER BY request_session_id ;

-- Look for blocking
SELECT  tl.resource_type ,
        tl.resource_database_id ,
        tl.resource_associated_entity_id ,
        tl.request_mode ,
        tl.request_session_id ,
        wt.blocking_session_id ,
        wt.wait_type ,
        wt.wait_duration_ms
FROM    sys.dm_tran_locks AS tl
        INNER JOIN sys.dm_os_waiting_tasks AS wt
           ON tl.lock_owner_address = wt.resource_address
ORDER BY wait_duration_ms DESC ;
```

Script 11: Identifying locking and blocking using sys.dm_tran_locks

The first query shows lock types and their status by SPID, filtered by the current database, and eliminating the current connection and the system SPIDs. The second query provides information regarding any blocking that may be occurring, instance-wide. Notice that this second query joins to the `sys.dm_os_waiting_tasks` DMV to obtain data on the length of time a process has been waiting, due to blocking, and on which resource. We'll cover this DMV in detail in the later section on "Operating System".

Unless you have pretty severe blocking issues, you'll typically need to run each of these queries multiple times to catch blocking. If you do identify two data modification statements, or a query and a data modification, which are embracing in severe blocking, or even deadlocks, then you'll need to extract the SQL text for the queries, examine them, run them on a test system, with Profiler tracing running, and work out a way to tune the queries, or add indexes, to alleviate the problem.

# Databases and Indexes

The DMOs covered in this section are targeted at the database/file level. By Microsoft, they are logically divided into two categories:

- **Database-related** – contains DMVs that allow us to investigate table and index page and row counts for a given database, as well as page allocation at the file level. It also provides a couple of DMVs dedicated to investigating usage of the `TempDB` database
- **Index-related** – contains DMVs specifically related to indexes, their characteristics, how they are used and to help identify potentially-useful indexes for your workload.

However, all of the views in these two categories begin with the name "`sys.dm_db_`". The main focus of this session is on DMVs that can help the DBA define an effective indexing strategy, since this is one of the best ways to ensure that the most significant and frequent queries are able to read the data they require in a logical, ordered fashion, and so avoid unnecessary I/O. Finding the correct balance between too many indexes and too few indexes, and having the "proper" set of indexes in place is extremely important for a DBA that wants to get the best performance from SQL Server.

We'll also include on script for monitoring the `TempDB` database. `TempDB` is a "global resource" used to store temporary data for user and internal objects, for all users connected to a given SQL Server instance. This includes, for example, internal worktables used to store results from cursors, and user objects such as temporary tables and table variables. If you use any of these heavily, space in this database can fill out surprisingly quickly.

Again, unless stated otherwise, all of the queries in this section work with SQL Server 2005, 2008, and 2008 R2 and all require `VIEW SERVER STATE` permission.

## DMV#11: Find Missing Indexes

In order to find out which indexes are potentially missing from a given database, we'll need to make use of three closely-related DMVs. The first one is **`sys.dm_db_missing_index_group_stats`**, which is described by BOL as follows:

> "Returns summary information about groups of missing indexes, excluding spatial indexes. Information returned by *`sys.dm_db_missing_index_group_stats`* is updated by every query execution, not by every query compilation or recompilation. Usage statistics are not persisted and are kept only until SQL Server is restarted. Database administrators should periodically make backup copies of the missing index information if they want to keep the usage statistics after server recycling."

The second DMV is **`sys.dm_db_missing_index_groups`**, which BOL describes as follows:

> "Returns information about what missing indexes are contained in a specific missing index group, excluding spatial indexes."

It is basically just a join table between `sys.dm_db_missing_index_group_stats` and our third DMV, which is **`sys.dm_db_missing_index_details`**, which BOL describes like this:

> "Returns detailed information about missing indexes, excluding spatial indexes."

By joining these three DMVs together, we get a very useful missing index query, as shown in Script 12.

```
-- Missing Indexes in current database by Index Advantage
SELECT  user_seeks * avg_total_user_cost * ( avg_user_impact * 0.01 )
                                                        AS [index_advantage]
,
        migs.last_user_seek ,
        mid.[statement] AS [Database.Schema.Table] ,
        mid.equality_columns ,
        mid.inequality_columns ,
        mid.included_columns ,
        migs.unique_compiles ,
        migs.user_seeks ,
        migs.avg_total_user_cost ,
        migs.avg_user_impact
FROM    sys.dm_db_missing_index_group_stats AS migs WITH ( NOLOCK )
        INNER JOIN sys.dm_db_missing_index_groups AS mig WITH ( NOLOCK )
           ON migs.group_handle = mig.index_group_handle
        INNER JOIN sys.dm_db_missing_index_details AS mid WITH ( NOLOCK )
           ON mig.index_handle = mid.index_handle
WHERE   mid.database_id = DB_ID()
ORDER BY index_advantage DESC ;
```

Script 12: Identifying potentially-useful indexes

This query uses the various statistics stored regarding data access patterns for a particular table, to calculate the possible "advantage" of adding a certain index. Indexes with a higher `index_advantage` are those that SQL Server considers will have the biggest positive impact on reducing workload, based on reduction of query cost and the projected number of times the index will be used.

Bear in mind that if you make a change to an index, for a given table, then all of the missing index statistics for that table are cleared out, and repopulated again over time. If you run this query shortly after an index change, it will probably, and inaccurately, inform you that there are no missing indexes for this table.

Even though I really like this query, and I have gotten some amazing results with it over the last few years, it does have some limitations that need to be considered when using it.

First, it does not always specify the best column order for an index. If there are multiple columns listed under `equality_columns` or `inequality_columns`, you will want to look at the selectivity of each of those columns within the equality and inequality results to determine the best column order for the prospective new index. Second, it does not consider filtered indexes, which are new for SQL Server 2008. Finally, it is overly eager to suggest included columns, and to suggest new indexes in general.

You should never just blindly add every index that this suggests, especially if you have an OLTP workload. Instead, you need to examine the results of the query carefully and manually filter out results that are not part of your regular workload. I always start by examining the `last_user_seek` column; if the `last_user_seek` time is a few days or even weeks ago, then the queries that caused SQL Server to want that index are probably from a random ad-hoc query, or part of a infrequently run report query. On the other hand, if the `last_user_seek` time was a few seconds or a few minutes ago, then it is probably part of your regular workload, and you should consider that possible index more carefully.

Regardless of what this query recommends, I always look at the existing indexes on a table, including their usage statistics (see DMV#12), before I consider making any changes. Remember, a more volatile table should generally have fewer indexes than a more static table. I generally start to get very hesitant to add a new index to a table (for an OLTP workload) if the table already has more than about five or six effective indexes.

Don't forget, the system stored procedure, `sp_helpindex`, does not show included column information. This means that you should either use a replacement, such as Kimberly Tripp's SP_HELPINDEX2, or simply script out the `CREATE INDEX` statement for your existing indexes.

## DMV#12: Interrogate Index Usage

One of the most useful DMVs in the Indexing category is **`sys.dm_db_index_usage_stats`**, which is described by BOL as follows:

Returns counts of different types of index operations and the time each type of operation was last performed. Every individual seek, scan, lookup, or update on the specified index by one query execution is counted as a use of that index and increments the corresponding counter in this view. Information is reported both for operations caused by user-submitted queries, and for operations caused by internally generated queries, such as scans for gathering statistics.

This DMV provides invaluable information regarding if and how often your indexes are being used, for both reads and writes, and we'll provide several scripts that interrogate this DMV to provide information on:

- The distribution of your workload across your defined indexes
- Indexes that are not accessed by your workload, and so are prime candidates for deletion
- Indexes with a large number of writes and zero, or few reads. These are also candidates for removal, after further investigation

The first of these three scripts (Script 13) will list all of your heap tables, clustered indexes, and non-clustered indexes, along with the number of reads, writes, and the fill factor for each index.

```
--- Index Read/Write stats (all tables in current DB)
SELECT  OBJECT_NAME(s.[object_id]) AS [ObjectName] ,
        i.name AS [IndexName] ,
        i.index_id ,
        user_seeks + user_scans + user_lookups AS [Reads] ,
        user_updates AS [Writes] ,
        i.type_desc AS [IndexType] ,
        i.fill_factor AS [FillFactor]
FROM    sys.dm_db_index_usage_stats AS s
        INNER JOIN sys.indexes AS i ON s.[object_id] = i.[object_id]
WHERE   OBJECTPROPERTY(s.[object_id], 'IsUserTable') = 1
        AND i.index_id = s.index_id
        AND s.database_id = DB_ID()
```

```
ORDER BY OBJECT_NAME(s.[object_id]) ,
        writes DESC ,
        reads DESC ;
```

Script 13: How are indexes being used?

This is a very useful query for better understanding your workload. You can use it to help determine how volatile a particular index is, and the ratio of reads to writes. This can help you refine and tune your indexing strategy. For example, if you had a table that was pretty static (very few writes on any of the indexes), you could feel more confident about adding more indexes that are listed in your missing index queries. If you have SQL Server 2008 Enterprise Edition, this query could help you decide whether it would be a good idea to enable data compression (either Page or Row). An index with very little write activity is likely to be a better candidate for data compression than an index that is more volatile.

The next script (Script 14) uses `sys.indexes` and `sys.objects` to find tables and indexes in the current database that do not show up in `sys.dm_db_index_usage_stats`. This means that these indexes have no reads or writes since SQL Server was last started, or since the current database was closed or detached, whichever is shorter.

```
-- List unused indexes
SELECT  OBJECT_NAME(i.[object_id]) AS [Table Name] ,
        i.name
FROM    sys.indexes AS i
        INNER JOIN sys.objects AS o ON i.[object_id] = o.[object_id]
WHERE   i.index_id NOT IN ( SELECT  s.index_id
                            FROM    sys.dm_db_index_usage_stats AS s
                            WHERE   s.[object_id] = i.[object_id]
                                    AND i.index_id = s.index_id
                                    AND database_id = DB_ID() )
        AND o.[type] = 'U'
ORDER BY OBJECT_NAME(i.[object_id]) ASC ;
```

Script 14: Finding unused indexes

If SQL Server has been running long enough that you have complete, representative workload, then there is a good chance that those indexes (and perhaps tables) are "dead", meaning they are no longer used by your database and can potentially be dropped, after doing some further investigation.

Our final `sys.dm_db_index_usage_stats` query filters by the current database, and only includes non-clustered indexes. It can help you decide whether the cost of maintaining a particular index outweighs the benefit you are receiving from having it in place.

```sql
-- Possible Bad NC Indexes (writes > reads)
SELECT  OBJECT_NAME(s.[object_id]) AS [Table Name] ,
        i.name AS [Index Name] ,
        i.index_id ,
        user_updates AS [Total Writes] ,
        user_seeks + user_scans + user_lookups AS [Total Reads] ,
        user_updates - ( user_seeks + user_scans + user_lookups )
            AS [Difference]
FROM    sys.dm_db_index_usage_stats AS s WITH ( NOLOCK )
        INNER JOIN sys.indexes AS i WITH ( NOLOCK )
            ON s.[object_id] = i.[object_id]
            AND i.index_id = s.index_id
WHERE   OBJECTPROPERTY(s.[object_id], 'IsUserTable') = 1
        AND s.database_id = DB_ID()
        AND user_updates > ( user_seeks + user_scans + user_lookups )
        AND i.index_id > 1
ORDER BY [Difference] DESC ,
        [Total Writes] DESC ,
        [Total Reads] ASC ;
```

Script 15: Finding rarely-used indexes

When I run this query, I look for any indexes that have large numbers of writes with zero reads. Any index that falls into that category is a pretty good candidate for deletion (after some further investigation). You want to make sure that your SQL Server instance has been running long enough that you have your complete, typical workload included. Don't forget about periodic, reporting workloads that might not show up in your day-to-day workload. Even though the indexes that facilitate such workloads will be infrequently used, their presence will be critical.

Next, I look at rows where there are large numbers of writes and a small number of reads. Dropping these indexes will be more of a judgment call, depending on the table and how familiar you are with your workload.

## DMV#13: Table Storage Stats (Pages and Row Counts)

This script uses the **sys.dm_db_partition_stats** DMV, which is described by BOL as follows:

"Returns page and row-count information for every partition in the current database."

Script 16 will show you which tables in the current database have the most rows.

```sql
-- Table and row count information
SELECT  OBJECT_NAME(ps.[object_id]) AS [TableName] ,
        i.name AS [IndexName] ,
        SUM(ps.row_count) AS [RowCount]
FROM    sys.dm_db_partition_stats AS ps
        INNER JOIN sys.indexes AS i ON i.[object_id] = ps.[object_id]
                                    AND i.index_id = ps.index_id
WHERE   i.type_desc IN ( 'CLUSTERED', 'HEAP' )
        AND i.[object_id] > 100
        AND OBJECT_SCHEMA_NAME(ps.[object_id]) <> 'sys'
GROUP BY ps.[object_id] ,
        i.name
ORDER BY SUM(ps.row_count) DESC ;
```

Script 16: Which tables have the most rows?

This is useful information to know, especially if you are considering adding an index to a table, or simply doing index maintenance on a table. If you know that a table has 500 million rows, rather than 500 thousand rows, then you might take a different course of action.

## DMV#14: Monitor TempDB

This script uses the **sys.dm_db_file_space_usage** DMV, which has the following terse description in BOL:

> "Returns space usage information for each file in the database."

The queries in Script 17 return various statistics relating to the TempDB database, including the number of free pages, space used by user and system objects, amount of free space available, and also the space allocated to the version store, when using SNAPSHOT isolation.

```
-- Get Free Space in TempDB
SELECT  SUM(unallocated_extent_page_count) AS [free pages] ,
        ( SUM(unallocated_extent_page_count) * 1.0 / 128 ) AS [free space
in MB]
FROM    sys.dm_db_file_space_usage ;

-- Quick TempDB Summary
SELECT SUM(user_object_reserved_page_count) * 8.192 AS [UserObjectsKB] ,
      SUM(internal_object_reserved_page_count) * 8.192 AS
[InternalObjectsKB] ,
      SUM(version_store_reserved_page_count) * 8.192 AS [VersonStoreKB] ,
      SUM(unallocated_extent_page_count) * 8.192 AS [FreeSpaceKB]
FROM    sys.dm_db_file_space_usage ;
```

Script 17: Monitoring TempDB growth

The load on `TempDB` is only set to increase, as use of `SNAPSHOT` isolation becomes more popular, so it is wise to monitor its growth, and to follow "best practices" advice, such as sizing it appropriately, and locating it on its own disk array, to avoid I/O contention issues.

# Disk I/O

The need to minimize logical and physical I/O has been discussed at several points in this book. The collection of I/O-related DMOs allows us to investigate, specifically, physical I/O that is taking place on our system, when data is written to and read from disk.

The DMOs in this category (all of which start with "`sys.dm_io_`") give us a picture of I/O explicitly from the point of view of the disk subsystem, showing us, for example, how the I/O is distributed across various files on the disk, places where I/O is becoming a bottleneck, resulting in I/O stalls, and so on. This information can help the DBA to optimize the disk subsystem architecture, and might be used as evidence for the DBA to request more/faster disk drives.

Some physical I/O is unavoidable, of course; data must be written to disk, the transaction log has to be written to for every insert, update, delete, and even for bulk operations. However, before jumping to the conclusion that more disk power is required, remember that there is much that can be done in terms of query tuning and indexing to minimize unnecessary logical and physical I/O. The I/O information derived from the DMOs covered in this section should be considered alongside other DMVs that reference IO performance in some manner, including:

- **`sys.dm_exec_query_stats`** – IO that a given query has cost over the times it had been executed
- **`sys.dm_exec_connections`** – IO that has taken place on that connection
- **`sys.dm_exec_sessions`** – IO that has taken place during that session
- **`sys.dm_os_workers`** – IO that is pending for a given worker thread

All of the queries in this section work with SQL Server 2005, 2008, and 2008 R2 and all require `VIEW SERVER STATE` permission.

## DMV#15: Investigate Disk Bottlenecks via I/O Stalls

The DMV we'll use here is **`sys.dm_io_virtual_file_stats`**, which is described by BOL as follows:

> "Returns I/O statistics for data and log files. This dynamic management view replaces the fn_virtualfilestats function."

This DMV accepts two arguments, which are `database_id` and `file_id`. You can specify `NULL` for either one, in which case, information on all of the databases and/or all of the files will be returned.

Note that this DMV is accumulative; in other words, the values in the data columns increment continuously, from the point when the server was last restarted. This means that you need to take a 'baseline' measurement followed by the actual measurement and then subtract the two, so that you can see where I/O is "accumulating".

Script 18 allows you to see the number of reads and writes on each data and log file, for every database running on the instance. It is sorted by average I/O stall time, in milliseconds.

```
-- Calculates average stalls per read, per write, and per total input/
output
-- for each database file.
SELECT  DB_NAME(database_id) AS [Database Name] ,
        file_id ,
        io_stall_read_ms ,
        num_of_reads ,
        CAST(io_stall_read_ms / ( 1.0 + num_of_reads ) AS NUMERIC(10, 1))
            AS [avg_read_stall_ms] ,
        io_stall_write_ms ,
        num_of_writes ,
        CAST(io_stall_write_ms / ( 1.0 + num_of_writes ) AS NUMERIC(10, 1))
            AS [avg_write_stall_ms] ,
        io_stall_read_ms + io_stall_write_ms AS [io_stalls] ,
        num_of_reads + num_of_writes AS [total_io] ,
        CAST(( io_stall_read_ms + io_stall_write_ms ) / ( 1.0 + num_of_
reads
                                                          + num_of_writes)
           AS NUMERIC(10,1)) AS [avg_io_stall_ms]
FROM    sys.dm_io_virtual_file_stats(NULL, NULL)
ORDER BY avg_io_stall_ms DESC ;
```

Script 18: Investigating I/O stalls

This query shows which files are waiting the most time for disk I/O and can help you to decide where to locate individual files based on the disk resources you have available. You can also use it to help persuade someone like a SAN engineer that SQL Server is seeing disk bottlenecks for certain files.

## DMV#16: Investigate Disk Bottlenecks via Pending I/O

Here, we take a slightly different approach to investigating disk I/O bottlenecks, using the **`sys.dm_io_pending_io_requests`** DMV, which is described, rather tersely, by BOL as follows:

> "Returns a row for each pending I/O request in SQL Server."

The data in the DMV provides a "point in time" snapshot of I/O requests that are pending on your system, right now.

```
-- Look at pending I/O requests by file
SELECT  DB_NAME(mf.database_id) AS [Database] ,
        mf.physical_name ,
        r.io_pending ,
        r.io_pending_ms_ticks ,
        r.io_type ,
        fs.num_of_reads ,
        fs.num_of_writes
FROM    sys.dm_io_pending_io_requests AS r
        INNER JOIN sys.dm_io_virtual_file_stats(NULL, NULL) AS fs
                                  ON r.io_handle = fs.file_handle
        INNER JOIN sys.master_files AS mf ON fs.database_id = mf.database_
id
                                     AND fs.file_id = mf.file_id
ORDER BY r.io_pending ,
        r.io_pending_ms_ticks DESC ;
```

Script 19: Investigating pending I/O requests

Since this data represents a point-in-time snapshot of activity, you would want to run this query multiple times to see if the same files (and drive letters) consistently show up at the top of the list. If that happens, then it is evidence of I/O bottlenecks for that file or drive letter. You could use this to help convince your SAN engineer that the system was experiencing I/O issues for a particular LUN.

The last two columns in the query return the cumulative number of read and writes for the file since SQL Server was started (or since the file was created, whichever was shorter). This information is helpful when trying to decide which RAID level to use for a particular drive letter. For example, files with more write activity will usually perform better on a RAID 10 LUN than they will on a RAID 5 LUN. Knowing the relative read/write ratio for each file can help you locate your database files on an appropriate LUN.

# Operating System

Now we are right down at the level of the operating system, and the "worker threads" that carry out the tasks required by our transactions. The DMOs that are provided in this category, all of which start with the name "`sys.dm_os_`", provide extremely detailed information about the way SQL Server is interacting with the OS and in turn the hardware. These DMOs can be used to answer such questions as:

- What kinds of things have the SQL Server OS threads have been waiting on? (**`wait_stats`**)
- What are the values of the SQL Server performance counters, and how are they decoded? (**`performance_counters`**)
- Is there currently a CPU utilization concern? (**`ring_buffers`**, **`os_schedulers`**, **`wait_stats`**)
- What are the characteristics of the machine that SQL Server is running on? (**`sys_info,`**)
- How is memory as a whole being utilized? (**`sys_memory`**, **`process_memory`**)
- How is the cache memory being utilized? (**`memory_cache_counters`**, **`buffer_descriptors`**)

All of the queries in this section work with SQL Server 2005, 2008, and 2008 R2 and all require `VIEW SERVER STATE` permission.

## A Brief Overview of SQL Server Waits

Arguably the most significant DMV in the "Operating System" category is `sys.dm_os_wait_stats`. Every time a session has to wait for some reason before the requested work can continue, SQL Server records the length of time waited, and the resource that is being waited on. The `sys.dm_os_wait_stats` DMV exposes these wait statistics, aggregated across all session IDs, to provide a summary review of where the major waits are on a given instance. This same DMV also exposes performance (perfmon) counters, which provide specific resource usage measurements (disk transfer rates, amount of CPU time consumed and so on). By correlating wait statistics with resource measurements, one can quickly locate the most 'contested resources' on your system, and so highlight potential bottlenecks.

***SQL Server 2005 Waits and Queues***

*The use of "waits and queues" as the basis for a performance tuning methodology is explained in an excellent whitepaper by Tom Davidson, which is available here: http://sqlcat.com/whitepapers/archive/2007/11/19/sql-server-2005-waits-and-queues.aspx*

Essentially, each request to SQL Server will result in the initiation of a number of "worker tasks". A SQL Server Scheduler assigns each task to a worker thread. Normally there is one SQLOS scheduler per CPU, and only one session per scheduler can be running at any time. It's the scheduler's job to spread the workload evenly between available worker threads. If a session's worker thread is running on the processor, the status of the session will be "Running", as exposed by the `Status` column of `sys.dm_exec_requests` DMV.
If a thread is "ready to go" but the scheduler to which it is assigned currently has another session running, then it will be placed in the "runnable" queue, which simply means it is in the queue to get on the processor. This is referred to as a **signal wait**. The signal wait time is exposed by the `signal_wait_time_ms` column, and is solely CPU wait time.
If a session is waiting for another resource to become available in order to proceed, such as a locked page, or if a running session needs to perform I/O, then it is moved to the wait list; this is a resource wait and the waiting session's status will be recorded as "suspended". The reason for the wait is recorded, and exposed in the `wait_type` column of the `sys.dm_os_wait_stats` DMV. The total time spent waiting is exposed by the `wait_time_ms` column, so the resource wait time can be calculated simply, as follows:

```
Resource waits = Total waits – Signal waits
               =(wait_time_ms) - (signal_wait_time_ms).
```

Signal waits are unavoidable in OLTP systems, comprising a large number of short transactions. The key metric, with regard to potential CPU pressure, is the signal wait as a percentage of the total waits. A high percentage signal is a sign of CPU pressure. The literature tends to quote "high" as more than about 25%, but it depends on your system. On my systems, I treat values higher than 10-15% as a worrying sign.

Overall, the use of wait statistics represents a very effective means to diagnose response times in your system. In very simple terms, you either work, or you wait:

```
Response time = service time + wait time
```

If response times are slow and you find no significant waits, or mainly signal waits, then you know you need to focus on CPU. If, instead, you find the response time is mainly comprised of time spent waiting for other resources (network, I/O etc.) then, again, you know exactly where to focus your tuning efforts.

***Taking the Guesswork out of Performance Profiling***

*Mario Broodbakker has written an excellent introductory series of articles into using wait event to diagnose performance problems, here: http://www.simple-talk.com/author/mario-broodbakker/*

## DMV#17: Why are we Waiting?

Our first script in the OS category uses the **`sys.dm_os_wait_stats`** DMV, which is described by BOL as follows:

"Returns information about all the waits encountered by threads that executed. You can use this aggregated view to diagnose performance issues with SQL Server and also with specific queries and batches."

The simple query shown in Script 20 calculates signal waits and resource waits as a percentage of the overall wait time, in order to diagnose potential CPU pressure.

```
-- Total waits are wait_time_ms (high signal waits indicates CPU pressure)
SELECT  CAST(100.0 * SUM(signal_wait_time_ms) / SUM(wait_time_ms)
                              AS NUMERIC(20,2)) AS [%signal (cpu) waits] ,
        CAST(100.0 * SUM(wait_time_ms - signal_wait_time_ms)
        / SUM(wait_time_ms) AS NUMERIC(20, 2)) AS [%resource waits]
FROM    sys.dm_os_wait_stats ;
```

Script 20: Is there any CPU pressure?

This query is useful to help confirm CPU pressure. Since Signal waits are time waiting for a CPU to service a thread, if you record total signal waits above roughly 10-15% then this is a pretty good indicator of CPU pressure. These wait stats are cumulative since SQL Server was last restarted so you need to know what your baseline value for signal waits is, and watch the trend over time.

You can manually clear out the wait statistics, without restarting the server, by issuing a `DBCC SQLPERF` command, as follows:

```
-- Clear Wait Stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR) ;
```

If your SQL Server instance has been running for quite a while, and you make a significant change, such as adding an important new index, then you should consider clearing the old wait stats. Otherwise, the old cumulative wait stats will mask whatever impact your change has on the wait times.

Our second example script, using the `sys.dm_os_wait_stats` DMV will help determine on which resources SQL Server is spending the most time waiting.

```
-- Isolate top waits for server instance since last restart
-- or statistics clear
WITH    Waits
      AS ( SELECT   wait_type ,
                    wait_time_ms / 1000. AS wait_time_s ,
                    100. * wait_time_ms / SUM(wait_time_ms) OVER ( ) AS pct
,
                    ROW_NUMBER() OVER ( ORDER BY wait_time_ms DESC ) AS rn
           FROM     sys.dm_os_wait_stats
           WHERE    wait_type NOT IN ( 'CLR_SEMAPHORE', 'LAZYWRITER_SLEEP',
                                       'RESOURCE_QUEUE', 'SLEEP_TASK',
                                       'SLEEP_SYSTEMTASK',
                                       'SQLTRACE_BUFFER_FLUSH', 'WAITFOR',
                                       'LOGMGR_QUEUE', 'CHECKPOINT_QUEUE',
                                       'REQUEST_FOR_DEADLOCK_SEARCH',
                                       'XE_TIMER_EVENT', 'BROKER_TO_FLUSH',
                                       'BROKER_TASK_STOP',
                                       'CLR_MANUAL_EVENT',
                                       'CLR_AUTO_EVENT',
                                       'DISPATCHER_QUEUE_SEMAPHORE',
                                       'FT_IFTS_SCHEDULER_IDLE_WAIT',
                                       'XE_DISPATCHER_WAIT',
                                       'XE_DISPATCHER_JOIN' )
         )
    SELECT  W1.wait_type ,
            CAST(W1.wait_time_s AS DECIMAL(12, 2)) AS wait_time_s ,
            CAST(W1.pct AS DECIMAL(12, 2)) AS pct ,
            CAST(SUM(W2.pct) AS DECIMAL(12, 2)) AS running_pct
    FROM    Waits AS W1
            INNER JOIN Waits AS W2 ON W2.rn <= W1.rn
    GROUP BY W1.rn ,
            W1.wait_type ,
            W1.wait_time_s ,
            W1.pct
    HAVING  SUM(W2.pct) - W1.pct < 95 ; -- percentage threshold
```

Script 21: Report on top resource waits

This script will help you locate the biggest bottleneck, at the instance level, allowing you to focus your tuning efforts on a particular type of problem. For example, if the top cumulative wait types are disk I/O related, then you would want to investigate this issue further using disk-related DMV queries (see DMVs#15 and 16) and PerfMon counters.

## DMV#18: Expose Performance Counters

The DMV that exposes the PerfMon counters is **`sys.dm_os_performance_counters`**, which is described by BOL as follows:

> "Returns a row per performance counter maintained by the server. For information about each performance counter, see Using SQL Server Objects."

This is a very useful DMV, but it can be very frustrating to work with. Depending on the value for `cntr_type` for a given row, you will have to go through some interesting 'gyrations' to get meaningful information from this DMV. It is a replacement for the old `sys.sysperfinfo` from SQL Server 2000.

An in-depth discussion of how to use the performance counters through this DMV is out-of-scope here, and is not well-documented online. The topic is covered in much more detail in the forthcoming book "Performance Tuning with SQL Server DMVs", by Tim Ford and Louis Davidson.

Script 22 allows you to investigate unusual conditions that are filling up your transaction log. It returns the recovery model, log reuse wait description, transaction log size, log space used, % of log used, compatibility level, and page verify option for each database on the current SQL Server instance.

```
-- Recovery model, log reuse wait description, log file size,
-- log usage size and compatibility level for all databases on instance
SELECT  db.[name] AS [Database Name] ,
        db.recovery_model_desc AS [Recovery Model] ,
        db.log_reuse_wait_desc AS [Log Reuse Wait Description] ,
        ls.cntr_value AS [Log Size (KB)] ,
        lu.cntr_value AS [Log Used (KB)] ,
        CAST(CAST(lu.cntr_value AS FLOAT) / CAST(ls.cntr_value AS FLOAT)
                AS DECIMAL(18,2)) * 100 AS [Log Used %] ,
        db.[compatibility_level] AS [DB Compatibility Level] ,
        db.page_verify_option_desc AS [Page Verify Option]
FROM    sys.databases AS db
        INNER JOIN sys.dm_os_performance_counters AS lu
                ON db.name = lu.instance_name
        INNER JOIN sys.dm_os_performance_counters AS ls
                ON db.name = ls.instance_name
WHERE   lu.counter_name LIKE 'Log File(s) Used Size (KB)%'
        AND ls.counter_name LIKE 'Log File(s) Size (KB)%' ;
```

Script 22:What is filling up the transaction log?

I like to run this query whenever I am evaluating an unfamiliar database server, and it is also more-generally useful from a monitoring perspective. For example, if your log reuse wait description is something unusual, such as `ACTIVE_TRANSACTION`, and your transaction log is 85% full, then I would want some alarm bells to be going off.

## DMV#19: Basic CPU Configuration

This script uses the **`sys.dm_os_sys_info`** DMV, which is described by BOL as follows:

"Returns a miscellaneous set of useful information about the computer, and about the resources available to and consumed by SQL Server."

That is not the best description that I ever read…but actions speak louder than words, and the queries in Script 23 demonstrate a good use of this DMV. It returns the number of physical and logical CPUs you have on your SQL Server instance. It also gives you the `hyperthread_ratio`, and the amount of physical RAM, along with the last SQL Server Start time.

```
-- Hardware information from SQL Server 2008
-- (Cannot distinguish between HT and multi-core)
SELECT  cpu_count AS [Logical CPU Count] ,
        hyperthread_ratio AS [Hyperthread Ratio] ,
        cpu_count / hyperthread_ratio AS [Physical CPU Count] ,
        physical_memory_in_bytes / 1048576 AS [Physical Memory (MB)] ,
        sqlserver_start_time
FROM    sys.dm_os_sys_info ;

-- Hardware information from SQL Server 2005
-- (Cannot distinguish between HT and multi-core)
SELECT  cpu_count AS [Logical CPU Count] ,
        hyperthread_ratio AS [Hyperthread Ratio] ,
        cpu_count / hyperthread_ratio AS [Physical CPU Count] ,
        physical_memory_in_bytes / 1048576 AS [Physical Memory (MB)]
FROM    sys.dm_os_sys_info ;
```

Script 23: CPU configuration

This query is very useful for finding out basic hardware information regarding your database server, especially if you work in an organization that does not give DBAs any direct access to their database servers.

The `hyperthread_ratio` column treats both multi-core and hyperthreading the same (which they are as far as the logical processor count goes), so it cannot tell the difference between a quad-core processor and a dual-core processor with hyperthreading enabled. In each case, these queries would report a `hyperthread_ratio` of 4.

## DMV#20: CPU Utilization History

This script uses the `sys.dm_os_ring_buffers` DMV, which according to BOL is

"Identified for informational purposes only. Not supported. Future compatibility is not guaranteed."

Ooh, that sounds scary! Well despite the warning and lack of documentation in BOL, there are multiple blog posts from Microsoft employees that show examples of using this DMV. The one shown in Script 24 returns the CPU utilization history over the last 30 minutes both in terms of CPU usage by the SQL Server process and total CPU usage by all other processes on your database server. This query only works on SQL Server 2008 and SQL Server 2008 R2

```
-- Get CPU Utilization History for last 30 minutes (in one minute
intervals)
-- This version works with SQL Server 2008 and SQL Server 2008 R2 only
DECLARE @ts_now BIGINT = ( SELECT   cpu_ticks / ( cpu_ticks / ms_ticks )
                           FROM     sys.dm_os_sys_info
                         ) ;
SELECT TOP ( 30 )
        SQLProcessUtilization AS [SQL Server Process CPU Utilization] ,
        SystemIdle AS [System Idle Process] ,
        100 - SystemIdle – SQLProcessUtilization
                  AS [Other Process CPU Utilization] ,
        DATEADD(ms, -1 * ( @ts_now - [timestamp] ), GETDATE()) AS [Event
Time]
FROM    ( SELECT    record.value('(./Record/@id)[1]', 'int')  AS record_id
,
                    record.value('(./Record/SchedulerMonitorEvent/
                                SystemHealth/SystemIdle)[1]','int')
                                                             AS
[SystemIdle] ,
                    record.value('(./Record/SchedulerMonitorEvent/
                                SystemHealth/ProcessUtilization)
[1]','int')
                                                   AS
[SQLProcessUtilization] ,
                    [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 N'%<SystemHealth>%'
                    ) AS x
        ) AS y
```

```
ORDER BY record_id DESC ;
```

Script 24: CPU utilization history

The query subtracts the `SystemIdle` value and the `SQL Server Process` value from 100 to arrive at the value for all other processes on the server. The results provide a handy way to see your recent CPU utilization history for the server as a whole, for SQL Server, and for other processes that are running on your database server (such as management software). Even though the granularity is only one minute, I like to be able to see this from T-SQL rather than having to look at PerfMon, or use WMI, to get CPU utilization information. In my experimentation, you can only retrieve 256 minutes worth of data from this query.

## DMV#21: Monitor Schedule activity

This script uses the **`sys.dm_os_schedulers`** DMV, which is described by BOL as follows:

> "Returns one row per scheduler in SQL Server where each scheduler is mapped to an individual processor. Use this view to monitor the condition of a scheduler or to identify runaway tasks."

The query in Script 25 will help detect blocking and can help detect as well as confirm CPU pressure.

```
-- Get Avg task count and Avg runnable task count
SELECT  AVG(current_tasks_count) AS [Avg Task Count] ,
        AVG(runnable_tasks_count) AS [Avg Runnable Task Count]
FROM    sys.dm_os_schedulers
WHERE   scheduler_id < 255
        AND [status] = 'VISIBLE ONLINE' ;
```

Script 25: Detect potential blocking via scheduler activity

High, sustained values for the current_tasks_count column usually indicate a blocking issue, and you can investigate this further using DMV#8. I have also seen it be a secondary indicator of I/O pressure, since high, sustained values for Avg Task Count can sometimes also be caused by I/O pressure. High, sustained values for the runnable_tasks_count column are usually a very good indicator of CPU pressure. By "high, sustained values", I mean anything above about 10-20 for most systems.

Another query I use quite often, against the DMV, is one that will tell me whether non-uniform memory access (NUMA) is enabled on a given SQL Server instance, as shown in Script 26.

```
-- Is NUMA enabled
SELECT  CASE COUNT(DISTINCT parent_node_id)
          WHEN 1 THEN 'NUMA disabled'
          ELSE 'NUMA enabled'
        END
FROM    sys.dm_os_schedulers
WHERE   parent_node_id <> 32 ;
```

Script 26: Is NUMA enabled?

AMD based servers have supported hardware based NUMA for several years, while Intel based Xeon servers, have added hardware based NUMA with the Xeon 5500, 5600, and 7500 series. There is also software based NUMA.

## DMV#22: System-wide Memory Usage

This script uses the **sys.dm_os_sys_memory** DMV, described by BOL as follows:

"Returns memory information from the operating system. SQL Server is bounded by, and responds to, external memory conditions at the operating system level and the physical limits of the underlying hardware. Determining the overall system state is an important part of evaluating SQL Server memory usage."

This DMV, as demonstrated in Script 27, can tell you how much physical memory you have and how much is available, as well as information about the page file. Finally, it tells you whether the operating system is signaling a low or high memory state. This DMV was added in SQL Server 2008 and so only works with SQL Server 2008, and 2008 R2.

```
-- Good basic information about memory amounts and state
-- SQL Server 2008 and 2008 R2 only
SELECT  total_physical_memory_kb ,
        available_physical_memory_kb ,
        total_page_file_kb ,
        available_page_file_kb ,
        system_memory_state_desc
FROM    sys.dm_os_sys_memory ;
```

Script 27: Available physical memory and page file space

I like to use this query to gather basic diagnostic memory information about an instance. If the operating system signals that available physical memory is low, then this can help confirm that SQL Server is under external memory pressure, and you can investigate this issue further.

## DMV#23: Detect Memory Pressure

The **`sys.dm_os_process_memory`** DMV is very useful when investigating possible memory pressure, and is described by BOL as follows:

"Most memory allocations that are attributed to the SQL Server process space are controlled through interfaces that allow for tracking and accounting of those allocations. However, memory allocations might be performed in the SQL Server address space that bypasses internal memory management routines. Values are obtained through calls to the base operating system. They are not manipulated by methods internal to SQL Server, except when it adjusts for locked or large page allocations. All returned values that indicate memory sizes are shown in kilobytes (KB). The column *total_virtual_address_space_reserved_kb* is a duplicate of *virtual_memory_in_bytes* from *sys.dm_os_sys_info*."

This DMV provides information from the point of view of memory being used by the SQL Server process. The query in Script 28 is about as simple as it gets, but nevertheless summarize some useful memory data for your SQL Serve instance. This script only works with SQL Server 2008, and 2008 R2.

```
-- SQL Server Process Address space info (SQL 2008 and 2008 R2 only)
--(shows whether locked pages is enabled, among other things)
SELECT  physical_memory_in_use_kb ,
        locked_page_allocations_kb ,
        page_fault_count ,
        memory_utilization_percentage ,
        available_commit_limit_kb ,
        process_physical_memory_low ,
        process_virtual_memory_low
FROM    sys.dm_os_process_memory ;
```

Script 28: Detect memory pressure

The query shows how much physical memory is in use by SQL Server, which is nice, since you cannot always, or even usually, believe Task Manager. It also shows whether you have "Locked Pages in Memory" enabled, indicated by a value for `locked_page_allocations_kb` of greater than zero.

Finally, it indicates whether the SQL Server process has been notified by the operating system that physical or virtual memory is low, at the OS level, meaning that SQL Server should try to trim its working set. This is explained in some more detail here: HTTP://SUPPORT.MICROSOFT.COM/KB/918483.

Just for reference, DBCC MEMORYSTATUS shows a superset of similar information, but it is more difficult to work with programmatically.

## DMV#24: Investigate Memory Usage Across all Caches

Our penultimate OS DMV is **sys.dm_os_memory_cache_counters**, which is described by BOL as follows:

> "Returns a snapshot of the health of a cache. sys.dm_os_memory_cache_counters provides run-time information about the cache entries allocated, their use, and the source of memory for the cache entries."

Essentially, this DMV provides a snapshot of cache-usage values, based on current reality. The column single_pages_kb is the amount of memory allocated via the single-page allocator. This refers to the 8-KB pages that are taken directly from the buffer pool, for the cache in question. The column multi_pages_kb is the amount of memory allocated by using the multiple-page allocator of the memory node. This memory is allocated outside the buffer pool and takes advantage of the virtual allocator of the memory nodes.

Script 29 investigates memory usage in the SQL Plans (SQLCP) and Object Plans (OBJCP) cache stores.

```
-- Look at the number of items in different parts of the cache
SELECT  name ,
        [type] ,
        entries_count ,
        single_pages_kb ,
        single_pages_in_use_kb ,
        multi_pages_kb ,
        multi_pages_in_use_kb
FROM    sys.dm_os_memory_cache_counters
WHERE   [type] = 'CACHESTORE_SQLCP'
        OR [type] = 'CACHESTORE_OBJCP'
ORDER BY multi_pages_kb DESC ;
```

Script 29: Investigate memory use in the SQLCP and OBJCP cache stores

A large number of `multi_pages_kb` for either of these cache types can lead to decreased performance on builds previous to SQL Server 2005 SP2 (Build 3042). Since SQL Server 2005 SP2 is no longer a supported service pack, this is yet another reason to get SQL Server 2005 SP3 (Build 4035), and hopefully SQL Server 2005 SP3 CU9 (Build 4294) applied.

## DMV#25: Investigate memory use in the Buffer Pool

The final DMV that we'll look at in this Operating System category is **`sys.dm_os_buffer_descriptors`**, which is described by BOL as follows:

> "Returns information about all the data pages that are currently in the SQL Server buffer pool. The output of this view can be used to determine the distribution of database pages in the buffer pool according to database, object, or type"

When a data page is read from disk, the page is copied into the SQL Server buffer pool and cached for reuse. Each cached data page has one buffer descriptor. Buffer descriptors uniquely identify each data page that is currently cached in an instance of SQL Server. The `sys.dm_os_buffer_descriptors` DMV returns cached pages for all user and system databases, including pages that are associated with the Resource database.

As demonstrated by Script 30, this DMV can tell how your buffer pool memory is being used, i.e., which databases and which indexes are using the most memory in the buffer pool.

```sql
-- Get total buffer usage by database
SELECT  DB_NAME(database_id) AS [Database Name] ,
        COUNT(*) * 8 / 1024.0 AS [Cached Size (MB)]
FROM    sys.dm_os_buffer_descriptors
WHERE   database_id > 4 -- exclude system databases
        AND database_id <> 32767 -- exclude ResourceDB
GROUP BY DB_NAME(database_id)
ORDER BY [Cached Size (MB)] DESC ;

-- Breaks down buffers by object (table, index) in the buffer pool
SELECT  OBJECT_NAME(p.[object_id]) AS [ObjectName] ,
        p.index_id ,
        COUNT(*) / 128 AS [Buffer size(MB)] ,
        COUNT(*) AS [Buffer_count]
FROM    sys.allocation_units AS a
        INNER JOIN sys.dm_os_buffer_descriptors
                AS b ON a.allocation_unit_id = b.allocation_unit_id
        INNER JOIN sys.partitions AS p ON a.container_id = p.hobt_id
WHERE   b.database_id = DB_ID()
        AND p.[object_id] > 100 -- exclude system objects
GROUP BY p.[object_id] ,
        p.index_id
ORDER BY buffer_count DESC ;
```

Script 30: Buffer pool usage by database

The first query rolls up buffer pool usage by database. It allows you to determine how much memory each database is using in the buffer pool. It could help you to decide how to deploy databases in a consolidation or scale-out effort.

The second query tells you which objects are using the most memory in your buffer pool and is filtered by the current database. It shows the table or indexed view name, the index ID (which will be zero for a heap table), and the amount of memory used in the buffer pool for that object. It is also a good way to see the effectiveness of data compression in SQL Server 2008 Enterprise Edition and SQL Server 2008 R2 Enterprise Edition.

# Other Useful DMVs

There are a large range of DMVs that are selectively useful, depending on which SQL Server features you are using. In this section, we'll cover just a few of these "miscellaneous" DMVs that we have found useful in our environments.

Unless stated otherwise, the queries in this section work with SQL Server 2005, 2008, and 2008 R2 and require `VIEW SERVER STATE` permission.

## DMV#26: Rooting out Unruly CLR Tasks

The DMV covered here, **`sys.dm_clr_tasks`**, is only relevant if you have enabled the CLR on your SQL Server instance, and you have at least one CLR assembly loaded in one of the user databases on the SQL Server instance. The `sys.dm_clr_tasks` DMV is described by BOL as follows:

> "Returns a row for all common language runtime (CLR) tasks that are currently running. A Transact-SQL batch that contains a reference to a CLR routine creates a separate task for execution of all the managed code in that batch. Multiple statements in the batch that require managed code execution use the same CLR task. The CLR task is responsible for maintaining objects and state pertaining to managed code execution, as well as the transitions between the instance of SQL Server and the common language runtime."

Script 31 will help you uncover any long-running, potentially-troublesome CLR tasks.

```
-- Find long running SQL/CLR tasks
SELECT  os.task_address ,
        os.[state] ,
        os.last_wait_type ,
        clr.[state] ,
        clr.forced_yield_count
```

```
FROM    sys.dm_os_workers AS os
        INNER JOIN sys.dm_clr_tasks AS clr
                    ON ( os.task_address = clr.sos_task_address )
WHERE   clr.[type] = 'E_TYPE_USER' ;
```

Script 31: Seeking out troublesome CLR routines

You want to be on the lookout for any rows that have a `forced_yield_count` above zero, or for rows that have a `last_wait_type` of `SQLCLR_QUANTUM_PUNISHMENT`. This portentously-named wait type indicates that the task previously exceeded its allowed quantum, and so caused the SQL OS scheduler to intervene and reschedule it at the end of the queue. The `forced_yield_count` indicates the number of times that this has happened.

If you see either of these, have a word with your developers about their CLR assemblies, and inform them that their CLR has been issued the soccer equivalent of a "yellow card", and that if they don't fix it, you'll be issuing the red one.

## DMV#27: Full Text Search

In order to take a peek at Full text catalogs, and how they are populated, we need to use two closely-related DMVs. The first is **`sys.dm_fts_active_catalogs`**, which is described by BOL as follows:

> "Returns information on the full-text catalogs that have some population activity in progress on the server."

The second one is **`sys.dm_fts_index_population`** which BOL describes as follows:

> "Returns information about the full-text index populations currently in progress."

By joining together these two DMVs, we get a very useful summary of what is happening with our full text catalogs, as shown in Script 32. Of course, this query is only useful if you are using full text search.

```
-- Get population status for all FT catalogs in the current database
SELECT  c.name ,
        c.[status] ,
        c.status_description ,
        OBJECT_NAME(p.table_id) AS [table_name] ,
        p.population_type_description ,
        p.is_clustered_index_scan ,
        p.status_description ,
        p.completion_type_description ,
        p.queued_population_type_description ,
        p.start_time ,
        p.range_count
FROM    sys.dm_fts_active_catalogs AS c
        INNER JOIN sys.dm_fts_index_population AS p
                      ON c.database_id = p.database_id
                       AND c.catalog_id = p.catalog_id
WHERE   c.database_id = DB_ID()
ORDER BY c.name ;
```

Script 32: Surveying full text search catalogs

In my experience, I have found that not too many DBAs seem to be using full text search in SQL Server, which is a pity because SQL Server 2008 has integrated full text search (iFTS) that is much easier to implement and maintain than the old version of full text search in SQL Server 2005. The new iFTS also performs much better than the 2005 version did, both for index creation and maintenance, and for full text searches.

## DMV#28: Page Repair attempts in Database Mirroring

A useful DMV for monitoring Database Mirroring is **.dm_db_mirroring_auto_page_repair**, which is described by BOL as follows:

> "Returns a row for every automatic page-repair attempt on any mirrored database on the server instance. This view contains rows for the latest automatic page-repair attempts on a given mirrored database, with a maximum of 100 rows per database. As soon as a database reaches the maximum, the row for its next automatic page-repair attempt replaces one of the existing entries."

SQL Server 2008 added a new feature to database mirroring called Automatic Page Repair. This feature allows either side of a database mirroring session to request data from the other side of the session that can be used to repair certain 823, 824, and 829 errors asynchronously. Paul Randal does a great job of explaining it in more detail here: HTTP://WWW.SQLSKILLS.COM/BLOGS/PAUL/POST/SQL-SERVER-2008-AUTOMATIC-PAGE-REPAIR-WITH-DATABASE-MIRRORING.ASPX.

Script 33 tells you whether you have had any automatic page repair attempts with SQL Server 2008 database mirroring, along with the results of the attempt. This DMV was added in SQL Server 2008 and so queries that use this DMV will only work with SQL Server 2008, and 2008 R2.

```
-- Check auto page repair history (New in SQL 2008)
SELECT  DB_NAME(database_id) AS [database_name] ,
        database_id ,
        file_id ,
        page_id ,
        error_type ,
        page_status ,
        modification_time
FROM    sys.dm_db_mirroring_auto_page_repair ;
```

Script 33: Checking for Auto Page Repair in mirrored databases

I like to periodically run this query on my SQL Server instances that have any mirrored databases to see if there have been any automatic page repair attempts since the instance was last restarted. Such attempts serve as an early warning sign of corruption issues that should be investigated further with `DBCC CHECKDB`.

# Conclusion

Ultimately, the exact methodology used to diagnose performance issues using the DMOs (alongside other tools) rather depends on the situation. Often, the DBA will be following a "treasure trail", starting with a specific reported issue (a session is blocked, reports are returning slowly, a file is filling up….), and trying to pull together a picture, in the form of a result set, that pinpoints the likely cause. This result set could report a list of blocking and blocked sessions, the top 5 longest running queries on the server, sessions causing the most IO, the 5 longest wait times being experienced on the server, or any number of other issues

The proactive DBA will regularly run monitoring scripts, such as those provided in this book, looking for warning signs of blocking, CPU/ IO / memory pressure, any changes in usage patterns that could affect response times, and attempting to resolve issues before they become serious problems.

Of course, unexpected issues do and will occur, and here too the DMOs can help. Used methodically, for example based on an analysis of the longest wait times on the system, they can lead you quickly and accurately to the most likely cause of the problem.

The DMOs don't make existing, built-in, performance tools obsolete. On the contrary, they complement these tools, but offer a flexibility, richness and granularity that are simply not available elsewhere. Furthermore, you don't need to master a new GUI or a new language in order to use them; it's all done in a language DBAs know and mostly love: T-SQL.

Hopefully, this book will help you get started quickly and smoothly!

## SQL Compare Pro® $595

Compare and synchronize SQL Server database schemas

- ↗ Automate database comparisons, and synchronize your databases
- ↗ Simple, easy to use, 100% accurate
- ↗ Save hours of tedious work, and eliminate manual scripting errors
- ↗ Work with live databases, snapshots, script files, or backups

**"SQL Compare and SQL Data Compare are the best purchases we've made in the .NET/ SQL environment. They've saved us hours of development time, and the fast, easy-to-use database comparison gives us maximum confidence that our migration scripts are correct. We rely on these products for every deployment."**

**Paul Tebbutt** Technical Lead, Universal Music Group

## SQL Data Compare Pro™ $595

Compare and synchronize SQL Server database contents

- ↗ Compare your database contents
- ↗ Automatically synchronize your data
- ↗ Row-level data restore
- ↗ Compare to scripts, backups, or live databases

**"We use SQL Data Compare daily and it has become an indispensable part of delivering our service to our customers. It has also streamlined our daily update process and cut back literally a good solid hour per day."**

**George Pantela** GPAnalysis.com

Visit **www.red-gate.com** for a 14-day, free trial

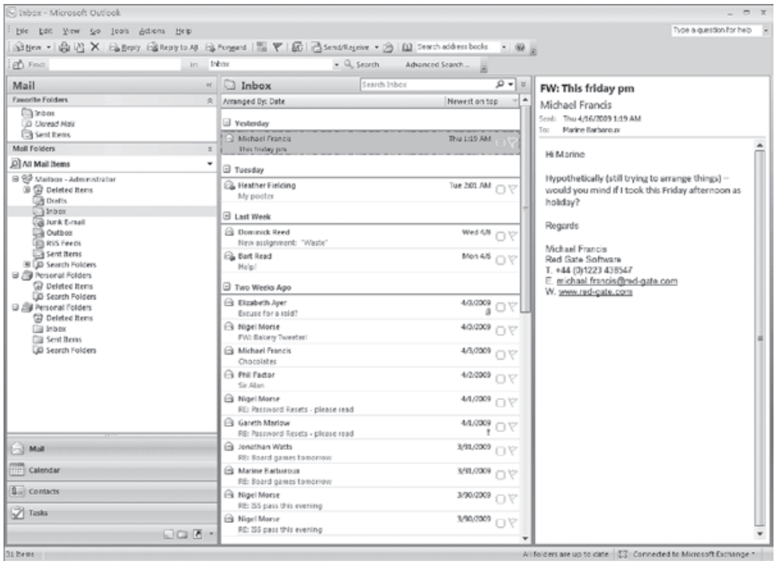
## Exchange Server Archiver

**$25** **per mailbox**
(200 mailboxes or over)

Email archiving software for Exchange

- Email archiving for Exchange Server
- Reduce size of information store – no more PSTs/mailbox quotas
- Archive only the mailboxes you want to
- Exchange, Outlook, and OWA 2003, 2007, and 2010 supported
- Transparent end-user experience – message preview, instantretrieval, and integrated search

**"Exchange Server Archiver is almost 100% invisible to Outlook end-users. The tool is simple to install and manage. This combined with the ability to set up different rules depending on user mailbox, makes the system easy to configure for all types of situations. I'd recommend this product to anyone who needs to archive exchange email."**
**Matthew Studer** Riverside Radiology Associates



Yesterday
Michael Francis — This friday pm — Thu 1:19 AM

Tuesday
Heather Fielding — My poster — Tue 2:01 AM

Last Week
Dominick Reed — New assignment: "Waste" — Wed 4/8
Bart Read — Help! — Mon 4/6

Two Weeks Ago
Elizabeth Ayer — Excuse for a raid? — 4/3/2009
Nigel Morse — FW: Bakery Tweeter! — 4/3/2009
Michael Francis — Chocolates — 4/3/2009
Phil Factor — Sir Alan — 4/2/2009
Nigel Morse — RE: Password Resets - please read — 4/1/2009
Gareth Marlow — RE: Password Resets - please read — 4/1/2009
Jonathan Watts — RE: Board games tomorrow — 3/31/2009
Marine Barbaroux — RE: Board games tomorrow — 3/31/2009
Nigel Morse — RE: ISS pass this evening — 3/30/2009
Nigel Morse — RE: ISS pass this evening — 3/30/2009

**FW: This friday pm**
Michael Francis
Sent: Thu 4/16/2009 1:19 AM
To: Marine Barbaroux

Hi Marine

Hypothetically (still trying to arrange things) – would you mind if I took this Friday afternoon as holiday?

Regards

Michael Francis
Red Gate Software
T. +44 (0)1223 438547
E. michael.francis@red-gate.com
W. www.red-gate.com

Visit **www.red-gate.com** for a 30-day, free trial