

SQL Server DMVs IN ACTION

Better queries with
Dynamic Management Views

Ian Stirk



 MANNING



**MEAP Edition
Manning Early Access Program
SQL Server DMVs in Action final version**

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Part 1: Starting the Journey

1. *Welcome to the Dynamic Management Views Goldmine*
2. *Common Patterns*

Part 2: DMV Discovery

3. *Index DMVs*
4. *Improving poor query performance*
5. *Further query improvements*
6. *Improving Server Level Performance*
7. *Common Language Runtime (CLR) DMVs*
8. *Resolving Transaction Issues*
9. *Database Level DMVs*

Part 3: Next Steps

10. *The self-healing database*
11. *Useful scripts*

1

The Dynamic Management Views Goldmine

This chapter covers

- What Dynamic Management Views are
- Why they are important
- Ready to run practical examples

Welcome to the world of Dynamic Management Views (DMVs). How would you like to fix problems on your SQL Servers with very little effort? Or fix problems before they become noticeable by irate users? Would you like to quickly discover the slowest SQL queries on your servers? Or get details of missing indexes that could significantly improve the performance of your queries? All these things and more are easily possible, typically in a matter of seconds, using DMVs.

In a nutshell, DMVs are views on internal SQL Server metadata, which can be used to significantly improve the performance of your SQL queries, often by an order of magnitude. A more thorough definition of DMVs follows on the next page.

The first part of fixing any problem is knowing what the underlying problem is. DMVs can give you precisely this information. DMVs will pinpoint where many of your problems are, often before they become painfully apparent.

DMVs are an integral part of Microsoft's flagship database SQL Server. Although they have existed since SQL Server 2005, their benefits are still relatively unknown, even by experienced software developers and Database Administrators (DBAs). Hopefully this book will help correct this deficit.

The aim of this book is to present and explain, in short snippets of prepackaged SQL, that can be used immediately, DMV queries that will give you a valuable insight into how your SQL Server and the queries running on it can be improved, often dramatically, quickly and easily.

In this chapter you will learn what DMVs are, the kinds of data they contain, and the types of problem DMVs can solve. I will outline the major groups the DMVs are divided into, and the ones we will be concentrating on. I will provide several example code snippets that you will find are immediately useful. DMVs will be discussed briefly in the context of other problem solving tools, and related structures (for example indexes and statistics).

I am sure after reading this chapter you will be pleasantly surprised when you discover the wealth of information that is available for free within SQL Server that can be accessed via DMVs, and the quite often impressive impact using this information can have. The DMV data is already out there waiting to be harvested, in so many ways it really is a goldmine!

1.1 What are Dynamic Management Views?

As queries run on a SQL Server database, SQL Server automatically records information about the activity that is taking place, internally into structures in memory, this information can be accessed via DMVs. So DMVs are basically SQL views on some pretty important internal memory structures.

Lots of different types of information is recorded which can be used for subsequent analysis, with the aim of improving performance, troubleshooting problems, or just gaining a better insight into how SQL Server works.

DMV information is stored on a per SQL Server instance level. We can however provide filtering to extract DMV data at varying levels of granularity, including: for a given database, table, or indeed a given query.

DMV information includes metrics that relate to indexes, query execution, the operating system, common language runtime (CLR), transactions, security, extended events, resource governor, service broker, replication, query notification, objects, input/output (I/O), full-text search, databases, database mirroring, change data capture (CDC), and much more. Additionally, there are many corollary areas that enhance and extend the DMV output, I will discuss these a little later, in the section entitled "DMV Companions".

Don't worry if you're not familiar with all these terms at the moment, the purpose of this book is to help explain them, present examples of how you can use them to improve the performance and understanding of your SQL queries and SQL Server itself.

Most sources categorize DMVs in the same manner that Microsoft has adopted, based on their area of functionality. We will use a similar approach in this book. A brief outline of each of the DMV categories follows.

Table 1.1 The major DMV groups

DMV Group	Description
Change Data Capture	Change Data Capture relates to how SQL Server captures change activity (inserts, updates, and deletes) across one or more tables, providing centralized processing. It can be thought of as a combination of trigger and auditing processing in a central area. These DMVs contain information relating to various aspects of Change Data Capture, including transactions, logging and errors. This occurs in SQL Server 2008 and higher.
Common Language Runtime	The Common Language Runtime allows non-database set processing code to be written in one of the .NET languages, offering a richer environment and language and often providing a magnitude increase in performance. These DMVs contain information relating to various aspects of the .NET Common Language Runtime, including application domains (these are wider in scope than a thread or smaller than a session), loaded assemblies, properties and running tasks.
Database Mirroring	The aim of database mirroring is to increase database availability. Transaction logs are moved quickly between servers, allowing fast fail over to the standby server. These DMVs contain information relating to various aspects of database mirroring, including connection information and page-repair details
Database	These DMVs contain information relating to various aspects of databases, including space usage, partition statistics, and session and task space information.
Execution	These DMVs contain information relating to various aspects of query

	execution, including cached plans, connections, cursors, plan attributes, stored procedure statistics, memory grants, query optimizer information, query statistics, active requests and sessions, SQL text, and trigger statistics.
Full-Text Search	Full-text search relates to the ability to search character based data using linguistic searches. This can be thought of as a higher level wildcard search. These DMVs contain information relating to various aspects of full-text search, including existing full-text catalogues, index populations currently occurring, and memory buffers/pools.
Index	These DMVs contain information relating to various aspects of indexes, including missing indexes, index usage (number of seeks, scans, lookups, by system or application, and when they last occurred), operational statistics (I/O, locking, latches, and access method), and physical statistics (size and fragmentation information).
Input/Output (IO)	These DMVs contain information relating to various aspects of I/O, including virtual file statistics (by database and file, number of reads/writes, amount of data read/written, and IO stall time), backup tape devices, and any pending IO requests.
Object	These DMVs contain information relating to various aspects of dynamic management objects, these relate to object dependencies.
Query Notification	These DMVs contain information relating to various aspects of query notification subscriptions in the server.
Replication	These DMVs contain information relating to various aspects of replication, including articles (type and status), transaction and schemas (table columns).
Resource Governor	In the past, running inappropriate ad-hoc queries on the database sometimes caused timeout and blocking problems. SQL Server 2008 implements a resource governor what controls the amount of resources different groups can have, allowing more controlled access to resources. These DMVs contain information relating to various aspects of resource governor, including resource pools, governor configuration and workload groups.
Service Broker	Service broker is concerned with providing transactional disconnected processing, allowing a wider range of architectural solutions to be created. These DMVs contain information relating to various aspects of service broker, including activated tasks, forwarded messages, connections, and queue monitors.
SQL Server Extended	Extended events allows SQL server integrate into Microsoft's wider event handling processes, allowing integration of SQL Server events with logging and monitoring tools. This occurs in SQL Server 2008 and higher.
SQL Server Operating System	These DMVs contain information relating to various aspects of the SQL Server Operating System (SQLOS), including performance counters, memory pools, schedulers, system information, tasks, threads, wait statistics, waiting tasks, and memory objects.
Transaction	These DMVs contain information relating to various aspects of transactions, including snapshot, database, session, and locks.
Security	These DMVs contain information relating to various aspects of

security, including audit actions, cryptographic algorithms supported, open cryptographic sessions, and database encryption state (and keys).

Since this book takes a look at DMVs from a practical, everyday trouble shooting and maintenance perspective, we will tend to concentrate on those DMVs that the DBA and database developer will use in help solve their everyday problems. With this in mind, we will concentrate on the following categories of DMV:

- Index
- Execution
- SQL Server Operating System
- Common Language Runtime
- Transaction
- Input/Output
- Database

If there is sufficient subsequent interest, perhaps another book could be written about the other DMVs groups.

1.1.1 A glimpse into SQL Server's internal data

As an example of what DMV information is captured, consider what happens when you run a query. An immense range of information is recorded, including:

- The query's cached plan (this describes at a low-level how the query is executed)
- What indexes were used
- What indexes the query would like to use but are missing
- How much IO occurred (both physical and logical)
- How much time was spent actually executing the query
- How much time was spent waiting on other resources
- What resources the query waiting upon

Being able to retrieve and analyze this information will not only give you a better understanding of how your query works, but will also allow you to produce better queries that take advantage of the available resources.

In addition to DMVs, there are several related functions that work in conjunction with DMVs, named Dynamic Management Functions (DMFs). In many ways DMFs are similar to standard SQL functions, being called repeatedly with a DMV supplied parameter. For example, the DMV `sys.dm_exec_query_stats` records details of the SQL being processed via a variable named `sql_handle`, if this `sql_handle` is passed as a parameter to the DMF `sys.dm_exec_sql_text`, the DMF will return the actual SQL text of the stored procedure or batch associated with this `sql_handle`.

All DMVs and DMFs belong to the `sys` schema, and when you reference them you must supply this schema name. The DMVs start with the signature of `sys.dm_*`, where the asterisk represents a particular subsystem. For example, to determine what requests are currently executing, we run the following:

```
SELECT * FROM sys.dm_exec_requests
```

Note, this query will give you raw details of the various requests that are currently running on your SQL Server, again don't worry if the output doesn't make much sense at the moment. I will provide much

more useful and understandable queries that use `sys.dm_exec_requests` later in the book, in the chapter related to Execution DMVs (chapter 5).

1.1.2 Aggregated results

The data shown via DMVs is accumulative, since the last SQL Server reboot or restart. Often this is useful, because we want to know the sum total effect for each of the queries that have run on the server instance or a given database.

However, if we are only interested in the actions of a given run of a query or batch, we can determine the effect of the query by taking a snapshot of the relevant DMV data, run our query, and then take another snapshot of the DMV data. Getting the delta between the two snapshots will provide us with details of the effect of the query that was run. An example of this approach is shown later, in the chapter concerning common patterns, section 2.10 entitled Calculating DMV changes.

1.1.3 Impact of running DMVs

Typically, when we query the DMVs to extract important diagnostic information, this querying has a minimal affect on the server and its resources. This is because the data is in memory and already calculated, we just need to retrieve it. To further reduce the impact of querying the DMVs, the sample code is typically prefixed with a statement that ignores locks and doesn't acquire any locks.

There are cases where the information isn't initially or readily available in the DMVs. In these cases, the impact of running the query may be significant. Luckily these DMVs are few in number and will be highlighted in the relevant section. One such DMV is used when calculating the degree of index fragmentation (`sys.dm_db_index_physical_stats`).

In summary, compared with other methods of obtaining similar information, for example by using the Database Tuning Advisor or SQL Server Profiler, using DMVs is relatively unobtrusive and has little impact on the system performance.

1.1.4 Part of SQL Server 2005 onwards

DMVs and DMFs have been an integral part of SQL Server since version 2005 onwards. In SQL Server 2005 there are 89 DMVs (and DMFs), and in SQL Server 2008 there are 136 DMVs. With this in mind, this book will concentrate on versions of SQL Server that are 2005 and higher. It is possible to discover the range of these DMVs by examining their names, by using the following query:

```
SELECT name, type_desc FROM sys.system_objects WHERE name LIKE 'dm_%' ORDER
BY name
```

In versions of SQL Server prior to SQL Server 2005, getting the level of detailed information given by DMVs, is either very difficult or impossible. For example, to obtain details of the slowest queries, we would typically have to run SQL Trace (this is the precursor of SQL Server Profiler) for a given duration and then spend often a considerable amount of time analyzing and aggregating the results. This being made more difficult since the parameters for the same queries would often differ. The corresponding work using DMVs can often be done in seconds.

1.2 The problems DMVs can solve

In the section entitled "What Are Dynamic Management Views", I briefly mentioned the different types of data that DMVs record. I can assure you this range is matched by depth too. DMVs allow you to view a great deal of internal SQL Server information that is a great starting point for determining the cause of a problem and provide potential solutions to fix many problems or give you a much better understanding of SQL Server and your queries.

The problems DMVs can solve can be grouped into Diagnosing Problems, Performance Tuning, and Monitoring. In the following sections we will discuss each of these in turn.

NOTE

DMVs are not the sole method of targeting the source of a problem or improving subsequent performance, but they can be used together with other tools to identify and correct concerns.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=639>

1.2.1 Diagnosing Problems

Diagnosing problems is concerned with identifying the underlying cause of the problem. This is perhaps the most common use of DMVs. It is possible to query the DMVs to diagnose many common problems including your slowest queries, the commonest causes of waiting/blocking, unused indexes, files having the most I/O, and lowest re-use of cached plans. Each of these areas of concern and more could be starting points to improving the performance of your SQL Server, whether you're a DBA maintaining a mature server environment or a developer working on a new project.

It is possible to view problem diagnosis at various levels, including from a server perspective, a database perspective or investigating a particular known troublesome query. Applying the correct filtering will allow us to use the DMVs at each of these levels.

Sometimes, identified problems are not real problems. For example, there may be queries that run very slowly but, they run at a time when it doesn't cause anyone any concern. So whilst we could fix them, it would be more appropriate to target our problem solving skills on problems that are deemed to be more important.

No one ever says their queries are running too fast, instead, users typically report how slow their queries seem to be running. Taking the slow running query as an example of a performance problem, we can use the DMVs to inspect the query's cached plan to determine how the query is accessing its data, how resources are being used (for example, if indexes are being used or table scans), if the statistics are out of date, identify any missing indexes, and target the particular statement or access path that is causing the slowness. Later we will look at interpreting the cached plan with a view to identifying performance bottlenecks.

Knowing the areas of the query that are slow allows you to try other techniques (for example, adding a new index) to see its affect on subsequent performance. Applying these new features leads us into the area of performance tuning. We will investigate a great many ways of identifying problems in the rest of the book.

One final point, sometimes if a query is too complicated, and contains lots of functionality, then try breaking it down into smaller steps, not only might this highlight the problem area with finer granularity, it might also solve it! Maybe the optimizer has more choices available to it with simpler queries and thus generates a better plan. We can see if this is the case by examining the relevant execution DMVs, as will become clear in chapter 5 relating to execution DMVs.

1.2.2 Performance Tuning

Performance tuning is concerned with applying suggested remedies to problems identified by problem diagnosis with a view to improving performance. Examination of the information shown by the DMVs should highlight areas where improvement can be made, for example, applying a missing index, removing contention/blocking, degree of fragmentation etc. Again the query's cached plan is a primary source of ideas for improvement.

Measurement of any improvement is typically reflected in time or IO counts, and can be made with traditional tools such as turning on STATISTICS IO or STATISTICS TIME SQL commands, or using a simple stopwatch. However, for more consistent results we can look at the time recording provided by the DMVs. This includes, for each individual SQL statement, time spent on the CPU (worker_time), and total time (elapsed_time). A large difference between these 2 times indicates a high degree of waiting/blocking may be occurring. Similarly DMVs also record the amount of IO (reads/writes at both the physical and logical level) that can be used to measure the effectiveness of a query, since less IO typically reflects a faster query.

Again we can examine the cached plan after the improvements have been made to determine if a more optimal access method has been chosen. Performance tuning is an iterative process. This new cached plan and DMV metrics could be used for further improvements, but again we need to ask if any remaining problem is worth solving, since we should always aim to fix what is deemed to be the most important problems first.

We need to be careful of the impact performance-based changes can have on the maintainability of systems, often these two needs are diametrically opposed since complexity is often increased. Rather than

guess where optimization is needed, appropriate tested should be undertaken first to determine where it is really needed. As the renowned computer scientist Donald Knuth said “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”. I will discuss this in more detail in chapter 5 concerning execution DMVs.

1.2.3 Monitoring

A large group of DMVs (those starting with sys.dm_exec_) relates to what is currently executing on the server. By repeatedly querying the relevant DMVs we get a view of the status of the server instance and also its history. Often this transient information is lost, but it is possible to store it for later analysis (for example into temporary or semi-permanent tables). An example of this is given in section 11.7 entitled “Who is doing what, and when?”

Sometimes you have problems with the overnight batch process, reported as timeout or slow running queries, and it would be nice to know what SQL is running during the time of this problem, giving you a starting point for further analysis.

Whilst you might know what stored procedure is currently running on your server (from your overnight batch scheduler or sp_who2), do you know what specific lines of SQL are executing? How are the SQL queries interacting? Is blocking occurring? We can get this information by using DMVs combined with a simple monitoring script. I have used such a script often to examine problems that occur during an overnight batch run.

NOTE

This example uses routines I have created and fully documented in the web links given in the code sample below (so you see, not only is code re-use good, but article re-use too). Rather than talk in detail about the contents of these two utilities, I'll talk about them as black boxes (if you do want to find out more about them, look here for the routine named dba_BlockTracer: <http://visualstudiomagazine.com/features/article.aspx?editorialid=2490> and here for the routine named dba_WhatSQLIsExecuting: <http://www.sqlservercentral.com/articles/DMV/64425/>). The code for both of these stored procedures is also available on the webpage for this book on the manning website. This way you'll be able to adapt this simple monitor pattern and possibly replace the 2 utilities with your own favorite utilities. Later in this chapter I will go through the code that forms the basis of one of stored procedures (dba_WhatSQLIsExecuting).

Listing 1.1 shows the code for a simple monitor.

Listing 1.1 A Simple Monitor

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

WAITFOR TIME '19:00:00'          #1
GO

PRINT GETDATE()
EXEC master.dbo.dba_BlockTracer    #2

IF @@ROWCOUNT > 0              #3
BEGIN
    SELECT GETDATE() AS TIME
    EXEC master.dbo.dba_WhatSQLIsExecuting #4
END

WAITFOR DELAY '00:00:15'        #5
GO 500                          #6

#1 Wait until 7pm
#2 Is anything blocked?
#3 If blocking occurring...
#4 Show SQL running
```

#5 Wait 15 seconds**#6 Repeat (500 times)**

The above code snippet waits until a specified time (7pm in the above example #1), and then prints the date/time and runs a routine named `dbo.dba_BlockTracer` #2. If anything is blocked, `dbo.dba_BlockTracer` displays information about both the blockers and the blocked items. Additionally, if anything is blocked (and output produced) the variable `@@ROWCOUNT` will have a non-zero value #3, this causes the output the date and time and list all the SQL that is running #4 (including the batch/stored procedure and the individual SQL statement within it that is running). The utility then waits a specified time (15 seconds in the above example #5) and repeats. All this is repeated (except waiting until 7pm) a number of times, as specified by the last `GO` statement (500 in the above example #6).

The routines show not only what is being blocked but also details of what SQL is running when any blocking occurs. When a SQL query runs, SQL Server assigns it a unique identifier, named the session id or the SQL Server process id (spid). You will notice the output in the various grids contain spids that can be used to link the output from the 2 utilities together.

An example of the type of output for this query is given in figure 1.1:

Root blocking spids	Owner	SQL Text	cpu	physical_io	DatabaseName	program_name	hostname	status	cmd	blocked
123	MARKET...	DECLARE @pCOB...	43469	9045	Paris	Microsoft Office 2003	FMDX3...	suspended	INSERT	0

Blocked spid	Blocked By	Owner	SQL Text	cpu	physical_io	DatabaseName	program_name	hostname	status	cmd
122	123	MARKET...	IF OBJECT_ID('CHK_dbo...	16	0	Paris	SQLAgent - TSQL Jo...	FMG-S3-0560	suspended	ALTER

TIME
2011-02-04 09:00:54.917

Spid	ecid	Database	User	Status	Wait	Individual Query	Parent Query	Program	Hostna...	nt_domain	start_time
85	0	Paris	SV...	running	NULL	SELECT StatM...	SELECT StatM...	SQLAgent - TSQL J...	FMG-S...	MARK...	2011-02-04 09:00:01.060
122	0	Paris	SV...	susp...	LCK...	ALTER TAB...	IF OBJECT_ID('...	SQLAgent - TSQL J...	FMG-S...	MARK...	2011-02-04 09:00:02.310
123	0	Paris	COO...	running	NULL	INSERT #M...	DECLARE @p...	Microsoft Office 2003	FMDX...	MARK...	2011-02-04 08:58:10.650
131	0	Paris	COO...	running	NULL	INSERT #M...	DECLARE @p...	Microsoft Office 2003	FMDX...	MARK...	2011-02-04 08:58:31.520
163	0	Paris	Ra...	susp...	WA...	WAITFOR D...	DROP TABLE...	Microsoft SQL Serv...	FMDX...	MARK...	2011-02-04 08:57:36.150

Figure 1.1 Output showing if anything is blocked, and what individual SQL queries are running

The first 2 grids show the root blocking spid (this is the cause of the blocking), and the blocked spid. This is followed by a grid showing the date and time the blocking occurred. Finally details of everything that is currently running is shown, this includes the individual line of SQL that is running together with the parent query (stored procedure or batch).

A special mention should be made about the humble `GO` command. The `GO` command will execute the batch of SQL statements that occurs after the last `GO` statements. If `GO` is followed by a number, then it will execute that number of times. This is useful in many circumstances, for example, after an `INSERT` statement if you put `GO 50`, the insert will occur 50 times.

This 'GO number' pattern can be extended to provide a simple concurrency/blocking/deadlock test harness. If a similar batch of SQL statements are entered into 2 or more distinct windows within SQL Server Management Studio (SSMS), and the statements are followed with a `GO 5000`, and all windows run at the same time, the effect of repeatedly running the SQL at the same time can be discovered.

It is of course possible to determine what is running irrespective of any blocking by using an even simpler monitoring query, given in snippet below:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

```
WAITFOR TIME '19:00:00'      #1
GO
```

```

SELECT GETDATE() AS TIME
EXEC master.dbo.dba_ WhatSQLIsExecuting #2

WAITFOR DELAY '00:00:15' #3
GO 500 #4

```

#1 Wait until 7pm
#2 Show running SQL
#3 Wait 15 seconds
#4 Repeat (500 times)

The query waits for a given time (7pm #1), and then displays the date and time together with details of what SQL queries are running #2, it then waits for a specified period (15 seconds #3) and repeats #4 (but does not wait until 7pm again!).

Queries often compete for resources, for example, exclusive access to a given set of rows in a table. This competition causes related queries to wait until the resource is free. This waiting affects performance. We can query the DMVs to determine what queries are waiting (being blocked) the most, and aim to improve them. We will identify the most blocked queries later, in chapter 4, section 4.5 “Queries that spend a long time being blocked”.

We can use the simple monitor utility discussed above to determine why these identified queries are being blocked, the DMVs will tell us what is blocked, but they don't identify what is blocking them. The monitoring utility can do this. The monitor utility can be a very powerful tool, in identifying why and how the most blocked queries are being blocked.

Having looked at what kind of problems DMVs can help solve, let's now dive into some simple but useful DMV example code that can be helpful in solving real life production problems.

1.3 DMV examples

The purpose of this section is to illustrate how easy it is to retrieve some really valuable information from SQL Server by querying the DMVs.

Don't worry if you don't understand all the details given in these queries immediately. I will not explain in detail here how the query performs its magic, after all this is meant to be a sample of what DMVs are capable of. I will however explain these queries fully later in the book.

NOTE

All the examples are prefixed with a statement concerning Isolation Level. This determines how the subsequent SQL statements in the batch interact with other running SQL statements. The statement sets the isolation level to read uncommitted. This ensures we can read data without waiting for locks to be released or acquiring locks ourselves, resulting in the query running more quickly with minimal impact on other running SQL queries. The statement used is:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

It is often the case that we have several different databases running on the same server. A consequence of this is, no matter how optimal your individual database may be, another database on the server, running suboptimally, may affect the server's resources and this may impact the performance of your database. Because of this, we offer scripts that inspect the DMVs across all the databases on the server. It is of course possible to target the queries to a specific database on the server instance by supplying a relevant WHERE clause (many other filters can be applied).

Bear in mind the purpose of these samples is to illustrate quickly how much useful information is freely and easily available within the DMVs. Richer versions of these routines will be provided later in the book.

1.3.1 Find your slowest queries

Does anyone ever complain “my queries are running too fast!” Almost without exception the opposite is the case, as queries are often reported as running too slowly. If you run the SQL query given in listing 1.2 you will identify the 20 slowest queries on your server.

Listing 1.2 Find your slowest queries

```

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20
CAST(total_elapsed_time / 1000000.0 AS DECIMAL(28, 2)) #1
AS [Total Elapsed Duration (s)]
, execution_count
, SUBSTRING (qt.text,(qs.statement_start_offset/2) + 1, #2
((CASE WHEN qs.statement_end_offset = -1
THEN LEN(CONVERT(NVARCHAR(MAX), qt.text)) * 2
ELSE
qs.statement_end_offset
END - qs.statement_start_offset)/2) + 1) AS [Individual Query]
, qt.text AS [Parent Query]
, DB_NAME(qt.dbid) AS DatabaseName
, qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) as qt
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
INNER JOIN sys.dm_exec_cached_plans as cp
on qs.plan_handle=cp.plan_handle
ORDER BY total_elapsed_time DESC #3

```

#1 Get query duration

#2 Extract SQL statement

#3 Sort by slowest queries

The DMV `sys.dm_exec_query_stats` contains details of various metrics that relate to an individual SQL statement (within a batch). These metrics include query duration #1 (`total_elapsed_time`), and the number of times the query has executed (`execution_count`). Additionally, it records details of the offsets of the individual query within the parent query. To get details of the parent query and the individual query #2, the offset parameters are passed to the DMF `sys.dm_exec_sql_text`. The Cross Apply statement can be thought of as a join to a table function that in this case takes a parameter. Here the parameter is the id of the cached plan that contains the textual representation of the query. The cached plan is a primary resource for discovering why the query is running slowly, and often will give an insight into how the query can be improved. The query's cached plan is output, as XML. The results are sorted by the `total_elapsed_time` #3. To limit the amount of output only the slowest 20 queries are reported on. Running the slowest queries query on my server gives the results shown in figure 1.2

	Total Elapsed Duration (s)	execution_count	Individual Query	Parent Query	DatabaseName	query_plan
1	1938.76	1	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>
2	1401.30	2	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>
3	1380.77	1	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>
4	710.39	2	update #AIPNL set /*D...	CREATE PROCEDUR...	PARISBAU	<ShowPlan>
5	698.82	1	select @COB as COB...	CREATE PROCEDUR...	PARISBAU	<ShowPlan>
6	629.58	1	INSERT INTO dbo.bgpt...	INSERT INTO dbo.bgp...	NULL	<ShowPlan>
7	620.99	2	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>
8	522.10	1	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>
9	411.12	1	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>
10	397.86	1	SELECT req [RequestD...	SELECT req [Request...	NULL	<ShowPlan>

Figure 1.2 Identify the slowest SQL queries on your server, sorted by duration

The results show the cumulative impact of individual queries, within a batch or stored procedure. Knowing the slowest queries will allow you to make targeted improvements, confident in the knowledge that any improvement to these queries will have the biggest impact on performance improvement.

It is possible to determine which queries are the slowest over a given time period by creating a snapshot of the relevant DMV data at the start and end of the time period, and calculating the delta. An

example of this is shown later, in the chapter concerning common patterns, in the section 2.10 entitled Calculating DMV changes.

The NULL values in the Databasename column mean the query was run either ad hoc or using prepared SQL (i.e. not as a stored procedure). This itself can be interesting since it indicates areas where stored procedures are not being re-used, and possible areas of security concern. Later, an improved version of this query will get the underlying database name for the ad-hoc or prepared SQL queries from another DMV source.

Slow queries can be a result of having incorrect or missing indexes, our next example will show how to discover these missing indexes.

1.3.2 Find those missing indexes

Indexes are a primary means of improving SQL performance. However, due to various reasons, for example, inexperienced developers or changing systems, useful indexes may not always have been created. Running the SQL query given in listing 1.3 will identify the top 20 indexes, ordered by impact (Total Cost), that are missing from your system.

Listing 1.3 Find those missing indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20
ROUND(avg_total_user_cost * avg_user_impact *      #1
      (user_seeks + user_scans),0) AS [Total Cost]
, avg_user_impact
, statement AS TableName
, equality_columns
, inequality_columns
, included_columns
FROM      sys.dm_db_missing_index_groups g
INNER JOIN sys.dm_db_missing_index_group_stats s
      ON s.group_handle = g.index_group_handle
INNER JOIN sys.dm_db_missing_index_details d
      ON d.index_handle = g.index_handle
ORDER BY [Total Cost] DESC                                #2
```

#1 Calculate cost

#2 Sort by cost

The DMV sys.dm_db_missing_index_group_stats contains metrics for missing indexes, including the how it would have been used (seek or scan), if it would have been used by an application or system (for example, DBCC), and various measures of cost saving by using this missing index. The DMV sys.dm_db_missing_index_details contains textual details of the missing index (what database/schema/table it applies to, what columns the index would include). These 2 DMVs (metrics and names) are linked together via another DMV sys.dm_db_missing_index_groups.

A note should be made of how the Total Cost of the missing index is calculated #1. Total cost should reflect the number of times the index would have been accessed (as a seek or scan), together with the impact of the index on its queries. The results are sorted by the calculated Total Cost #2.

Applying these indexes to your systems may have a significant impact on the performance of your queries.

Running the Missing Indexes query on my server gives the results shown in figure 1.3

	Total Cost	avg_user_impact	TableName	EqualityUsage	InequalityUsage	Include Columns
1	3846455	93.17	[Paris].[dbo].[RiskValue]	[RequestId]	NULL	[PositionGridCellId], [Co...
2	921788	34.19	[ParisDev].[dbo].[PNLValue]	[BaseValue]	[LocalValue]	[PNLValueId], [Position...
3	887350	99.76	[ParisDev].[dbo].[RequestPNL...	[BatchNbr]	NULL	NULL
4	620383	79	[ParisDev].[dbo].[PNLAdjustme...	[Status]	NULL	[PNLAdjustmentQueue...
5	282331	20.41	[ParisDev].[dbo].[PNLAdjustme...	NULL	[Status]	[PNLAdjustmentQueue...
6	249135	53.88	[ParisDev].[dbo].[Component]	[ComponentCode]	[BookCode]	[DealId]
7	231713	98.69	[ParisDev].[dbo].[RequestDeal...	[BatchNbr]	NULL	NULL
8	171123	14.43	[Paris].[dbo].[Deal]	[DealTypeId]	NULL	[DealId], [DealCode], [...]
9	150870	41.97	[ParisDev].[dbo].[PNLAdjustme...	NULL	[Status]	[PNLAdjustmentQueue...
10	73727	12.98	[ParisDev].[dbo].[PNLAdjustme...	NULL	[LocalValue], [...]	[PNLAdjustmentQueue...

Figure 1.3 Output from the Identify Missing Indexes SQL

The results show the most important missing indexes as determined by this particular method of calculating their Total Cost. You can see the database/schema/table that the missing index should be applied to. The other output columns relate to how the columns that would form the missing index would have been used by various queries, such as, if the columns have been used in equality or inequality clauses on the SQL WHERE statement. The last column lists any additional columns the missing index would like included at the leaf level for quicker access.

Given the importance of indexes to query performance, many aspects of index usage will be discussed throughout this book, and especially in chapter 3 entitled Index DMVs.

1.3.3 Identify what SQL statements are running now

Often you may know that a particular batch of SQL (or stored procedure) is running, but do you know how far it has got within the batch of SQL? This is particularly troublesome when the query seems to be running slowly or you want to ensure a particular point within the batch has safely passed.

Inspecting the relevant DMVs will allow you to see the individual SQL statements within a batch that are currently executing on your server.

To identify the SQL statements currently running now on your SQL Server, run the query given in listing 1.4. If a stored procedure or batch of SQL is running, the column Parent Query will contain the text of the stored procedure or batch, and the column Individual Query will contain the current SQL statement within the batch that is being executed (this can be used to monitor progress of a batch of SQL). Note if the batch contains only a single SQL statement, then this value is reported in both the Individual Query and Parent Query columns. Looking at the WHERE clause you will see we ignore any system processes (having a spid of 50 or less), and we also ignore this actual script too.

Listing 1.4 identify what SQL is running now

```

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT session_Id AS [Spid]
, ecid
, DB_NAME(sp.dbid) AS [Database]
, nt_username
, er.status
, wait_type
, SUBSTRING (qt.text, (er.statement_start_offset/2) + 1, #1
((CASE WHEN er.statement_end_offset = -1
      THEN LEN(CONVERT(NVARCHAR(MAX), qt.text)) * 2
      ELSE er.statement_end_offset
END - er.statement_start_offset)/2) + 1) AS [Individual Query]
, qt.text AS [Parent Query]
, program_name
, Hostname
, nt_domain
, start_time

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=639>


```

FROM sys.dm_exec_requests er          #2
INNER JOIN sys.sysprocesses sp ON er.session_id = sp.spid
CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) as qt
WHERE session_id > 50
AND session_id NOT IN (@@SPID)
ORDER BY session_id, ecid

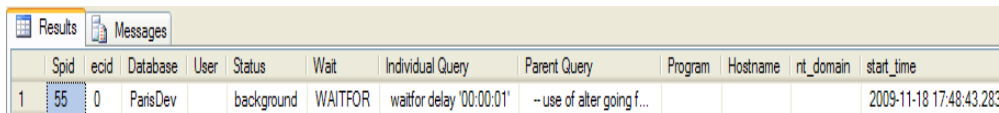
```

#1 Extract SQL statement

#2 Join request to sysprocesses

The DMV `sys.dm_exec_requests` contains details of each request, the SQL query #1, executing on SQL Server. This DMV is joined to the catalog view `sys.sysprocesses` #2 based on their session id. Catalog views are similar to DMVs but contain static data, I will talk more about them shortly, in the section entitled DMV companions. The catalog view `sys.sysprocesses` contains information about the environment from which the request originated, includes such details as user name, and the name of host it's running from. Combining the DMV and catalog view gives us a great deal of useful information about the queries that are currently running.

As discussed previously, in the section "Find your slowest queries", we get the running query's SQL text by passing the request's `sql_handle` to the DMF `sys.dm_exec_sql_text`, and apply string manipulation to that SQL text, to obtain the exact SQL statement that is currently running. Running the What SQL Is Running Now query on my server gives the results shown in figure 1.4



	Spid	ecid	Database	User	Status	Wait	Individual Query	Parent Query	Program	Hostname	nt_domain	start_time
1	55	0	ParisDev		background	WAITFOR	waitfor delay '00:00:01'	~ use of alter going f...				2009-11-18 17:48:43.283

Figure 1.4 Output identify what SQL queries are currently running on the server

The output shows the spid (process identifier), the ecid (this is similar to a thread within the same spid, and is useful for identifying queries running in parallel), the database name, the user running the SQL, the status (if the SQL is running or waiting), the Wait status (why it is waiting), the hostname, the domain name, and the start_time (useful for determining how long the batch has been running). These columns and their relevance will be explained in detail later in the book, in chapter 5, section 5.9, entitled "Current running queries"

The route a SQL query takes in answering a query can be seen by examining the query's cached plan, this can provide several clues as to why a query is performing as it is. Next we'll look at how these plans can be found quickly.

1.3.4 Quickly find a cache plan

The cached plan (execution plan) is a great tool for determining why something is happening, such as why a query is running slowly or if an index is being used. When a SQL query is run, it is first analyzed to determine what features, for example indexes, should be used to satisfy the query. Caching this access plan enables other similar queries (with different parameter values) to save time by re-using this plan.

It is possible to obtain the estimated or actual Execution Plan for a batch of SQL by clicking on the relevant icon in SQL Server Management Studio (SSMS). Typically the estimated plan differs from the actual plan in that the former is not actually run. The latter will provide details of actual row counts as opposed to estimated row counts (the discrepancy between the two row counts can be useful in determining if the statistics need to be updated).

However, there are problems with this approach. It may not be viable to run the query since it may be difficult to obtain (for example, the query takes too long to execute, after all that's often the reason we're looking at it!).

Luckily, if the query has been executed at least once already, it should exist as a cached plan, so we just need the relevant SQL to retrieve it using the DMVs. If you run the SQL query given in listing 1.5 you can retrieve any existing cached plans that contain the text given by the WHERE statement. In this case, the query will retrieve any cached plans that contain the text 'CREATE PROCEDURE' #1, of which there

should be many. Note you will need to enter some text that uniquely identifies your SQL, for example the stored procedure name, to retrieve the specific cached plans you would like to see.

Listing 1.5 Quickly find a cached plan

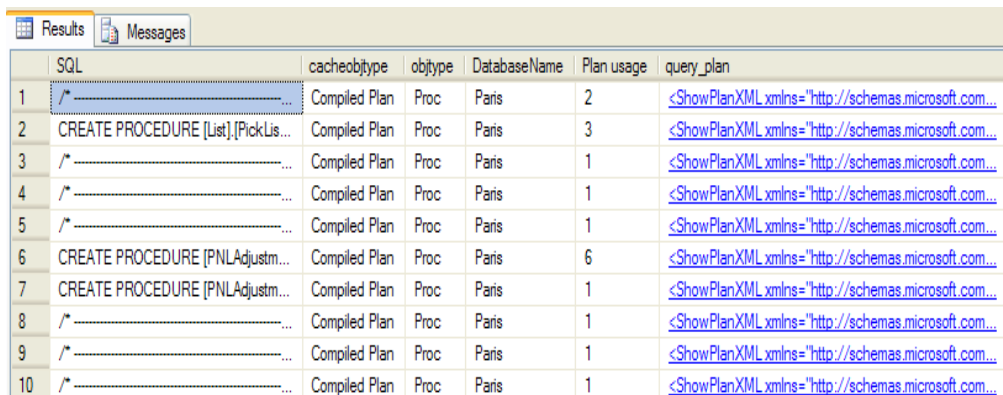
```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20
    st.text AS [SQL]
    , cp.cacheobjtype
    , cp.objtype
    , COALESCE(DB_NAME(st.dbid),
        DB_NAME(CAST(pa.value AS INT))+ '*' ,
        'Resource') AS [DatabaseName]
    , cp.usecounts AS [Plan usage]
    , qp.query_plan
FROM sys.dm_exec_cached_plans cp          #A
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp
OUTER APPLY sys.dm_exec_plan_attributes(cp.plan_handle) pa
WHERE pa.attribute = 'dbid'
AND st.text LIKE '%CREATE PROCEDURE%'    #1
```

#A Join cached plan and SQL text DMVs

#1 Text to search plan for

Running the Quickly Find A Cached Plan query on my server gives the results shown in figure 1.5.



	SQL	cacheobjtype	objtype	DatabaseName	Plan usage	query_plan
1	/*-----...	Compiled Plan	Proc	Paris	2	<ShowPlanXML xmlns="http://schemas.microsoft.com...
2	CREATE PROCEDURE [List].[PickLis...	Compiled Plan	Proc	Paris	3	<ShowPlanXML xmlns="http://schemas.microsoft.com...
3	/*-----...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...
4	/*-----...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...
5	/*-----...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...
6	CREATE PROCEDURE [PNLAdjustm...	Compiled Plan	Proc	Paris	6	<ShowPlanXML xmlns="http://schemas.microsoft.com...
7	CREATE PROCEDURE [PNLAdjustm...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...
8	/*-----...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...
9	/*-----...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...
10	/*-----...	Compiled Plan	Proc	Paris	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...

Figure 1.5 Output showing searched for cached plans

When you identify the relevant query you want the cached plan for (the above query is quite generic, looking for plans that contain the text 'CREATE PROCEDURE'), clicking on the column named query_plan will display the query plan. How it does this differs depending on if you're using SQL Server version 2005 or 2008. If you're using version 2005, clicking on the column will open up a new window showing the cached plan in XML format, if you save this XML with an extension of .sqlplan, and then open it separately (just double click it in Windows explorer), it will open showing a full graphical version of the plan in SSMS. If you're using SQL Server 2008, clicking on the query_plan column will open up the cached plan as a full graphical version, this is shown in figure 1.6.

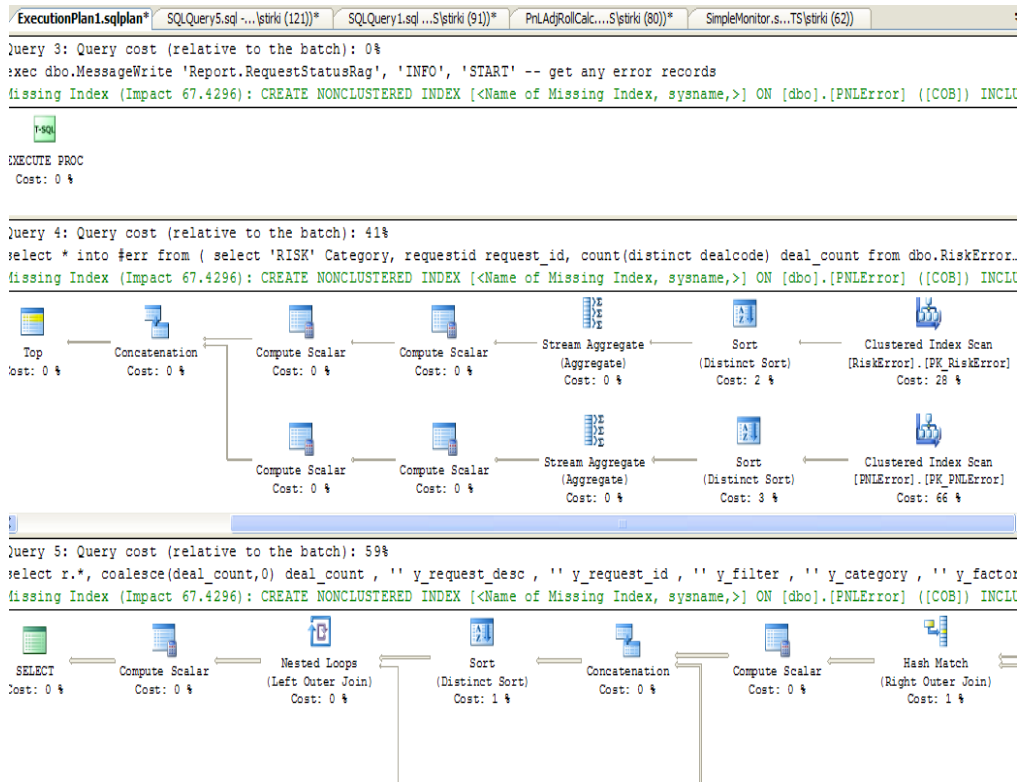


Figure 1.6 Missing Indexes details included with a 2008 Cached Plan

As a side note, if you're using SQL Server 2008, when you see the graphical version of the cached plan, if there are any missing indexes, they are given at the top of each section in green, with text starting "Missing Index" (again see figure 1.6), if you right mouse click on the diagram you can select the "missing index details...", clicking on this will open a new window with definition of the missing index, ready to add, you just need to add an appropriate index name. An example of this is shown in listing 1.6

Listing 1.6 Missing index details

```

/*
Missing Index Details from ExecutionPlan1.sqlplan
The Query Processor estimates that implementing the following index could improve the
query cost by 67.4296%.
*/

/*
USE [YourDatabaseName]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [dbo].[PNLError] ([COB])
INCLUDE ([RequestId],[DealCode])
GO
*/

```

If I search for the cached plan of a routine that contains a reference to something named SwapsDailyfile, I can quickly get its cached plan, part of which is shown below in figure 1.7

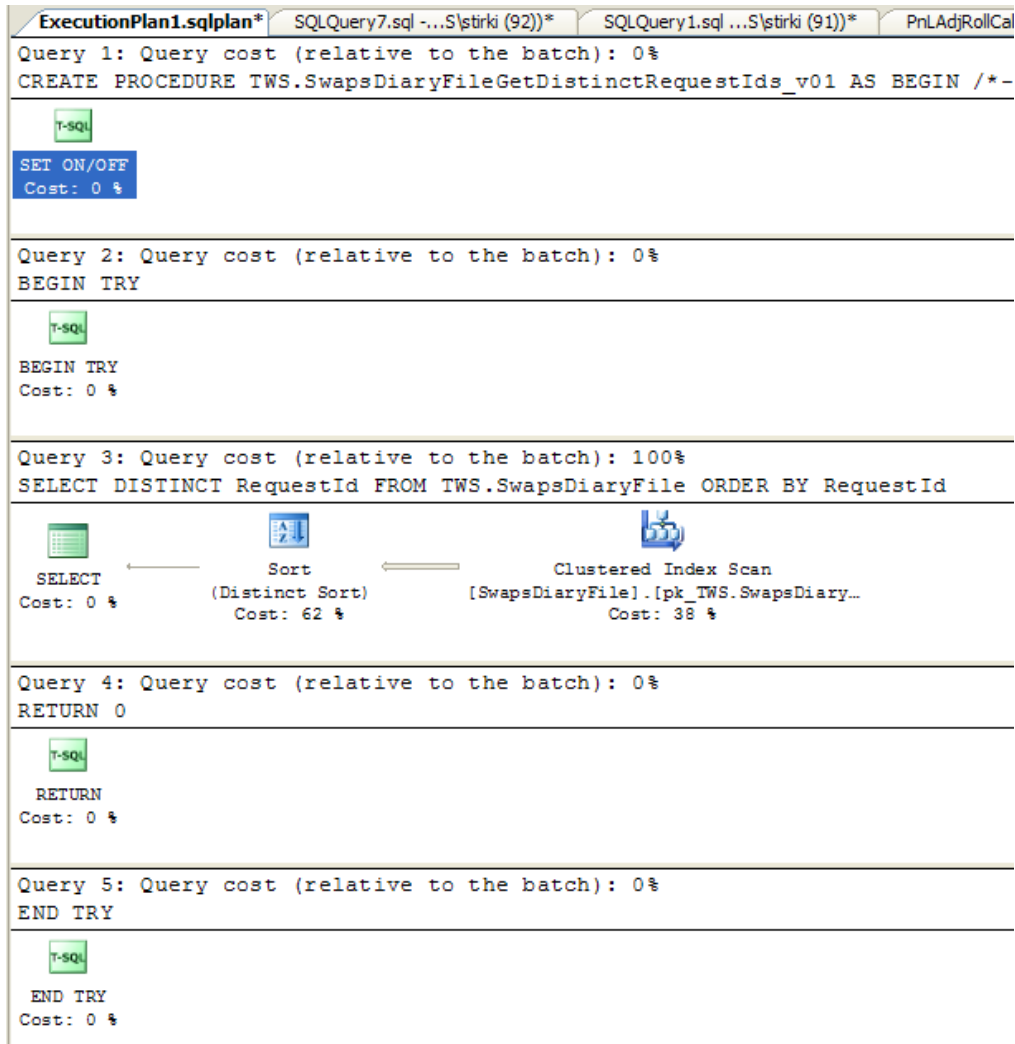


Figure 1.7 Cached plan showing cost by statement and within each statement

Looking at figure 1.7, you can see that each statement within a batch or stored procedure will have a query cost associated with it (here you'll notice the first 2 queries have a 0% cost, followed by another query that has a 100% cost. Once you find the section of code that has a high query cost, you should then inspect the components (shown as icons) that make up that cost. They too are numbered (in our example, Query 3 is divided into 3 parts, with cost values of 0%, 62% and 38%. You can thus identify the section within the batch that should be targeted for improvement, in our case is it the operation that cost 62%.

Before DMVs can be used, we need to ensure the appropriate permissions have been assigned, and we are aware of the accumulative nature of DMVs, these are discussed next.

1.4 Preparing to use DMVs

A great many exciting DMV insights into our SQL queries await us. But before we can dive into using DMVs, we need to ensure we are aware of certain prerequisites. The first of these relates to permissions to the DMVs/DMFs, and the second relates to circumstances under which you might want to clear the DMVs.

1.4.1 Permissions

There are two levels of DMVs and DMFs usage, namely server-scoped and database-scoped. Server-scoped require VIEW SERVER STATE permission on the server and database-scoped require VIEW

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=639>

DATABASE STATE permission on the database. Granting VIEW SERVER STATE permission infers VIEW DATABASE STATE permission on all the databases on the server.

Note that if a user has been granted VIEW SERVER STATE permission but has been denied VIEW DATABASE STATE permission, the user can see server level information but not database level information for the denied database.

If you're undertaking testing, you may want to be able to clear various DMVs, to do this you will need ALTER SERVER STATE on the server. Details of the various methods used to clear the DMVs are given in the following section.

Rather than assign these permissions to existing groups or users, it is often preferable to create a specific login (named, for example, DMV_Viewer) and assign appropriate permissions to that login.

1.4.2 Clearing DMVs

Often when we want to find the effect of a given query or system action, we will want to clear the relevant DMVs to give us a clean starting point from which to make measurements. Various DMVs can be cleared in different ways.

Clearing DMVs

Please note clearing DMVs on production machines should only be done after careful consideration, since it will result in queries taking longer to run because they need to be compiled again.

Before we discuss how the DMVs can be cleared, it is worth noting another approach exists to allow us to determine the effect of a given query or system action. DMV data is accumulative, so to determine the effect of a query, we can take a snapshot of the relevant DMVs, run the query under investigation, and then take a second snapshot. The effect of the query can be determined by comparing the two snapshots and calculating the differences (delta). Several examples of this approach are given subsequent chapters, for example, see section 6.8 entitled "Effect of running SQL queries on the performance counters".

Another possible way of using the DMVs without clearing them, is to retrieve SQL query data from a given time period (best if it has run recently), since several DMVs record when the query was last run.

The simplest way to clear the DMVs is to stop and restart the SQL Server service or alternatively reboot the SQL Server box. Whilst this may be the easiest method, it is probably also the most drastic in terms of impact on users, so it should be used with caution.

Alternatively, it is possible to clear some specific DMVs, in particular, those that relate to cached plans and query performance. These DMVs start with a signature of sys.dm_exec_.

To clear the DMVs that relate to cached plans, at the server level, we use the following command: DBCC FREEPROCCACHE. This clears all the cached plans on all databases on the server. In SQL Server 2008 this command can also be supplied with a parameter to remove a specific cached plan from the pool of cached plans.

The parameter supplied to DBCC FREEPROCCACHE is either a plan_handle, sql_handle or pool_name. plan_handle and sql_handle are 64bit identifiers of a query plan and batch of SQL statements respectively that are found in various DMVs. The pool_name is the name of a Resource Governor workload group within a resource pool.

We can also clear the cached plans for a specific database only, we can use the below:

```
DECLARE @DB_ID INT

SET @DB_ID = DB_ID('NameOfDatabaseToClear') -- Change this to your DB

DBCC FLUSHPROCINDB(@DB_ID)
```

When SQL Server is closed down or the SQL Server service is stopped, the DMV data is lost. There are of course methods of creating a more permanent version of this transient information for later analysis. An example of this is given in section 11.7 entitled "Who is doing what, and when?"

NOTE

It should be noted that not all queries are cached, these include DBCC commands and index reorganizations, additionally, queries can be removed from the cache when there are memory pressures.

The power of DMVs can be enhanced considerably if we link to various other database objects including indexes, statistics, and catalogued views. These are discussed next.

1.5 DMV Companions

Whilst this book is primarily concerned with the usage of DMVs, to get the most out of the DMVs it is necessary to know more about some peripheral but related areas, including catalog views, cached plans, indexes, and database statistics. Knowing about these other areas should give us a better understanding of what the DMVs provide and what can be done to improve performance.

1.5.1 Catalog views

DMVs and catalog views together provide a more complete view of your internal SQL Server data, where DMVs contain dynamic data and catalog views contain static data. To take full advantage of the various DMVs we will need to join the DMVs with various catalog views to produce more meaningful output. For example, when we are processing DMVs that relate to indexes, we will often join with the catalog view `sys.indexes` which contains information about indexes, such as index type or its uniqueness, created on tables in the current database.

Earlier versions of SQL Server held this internal meta data in system tables, these tables are still present in later versions. However it is not recommended to query these system tables directly, since future internal changes that Microsoft make may break your code. These system tables have been replaced by:

- Catalog views – tables that describe objects e.g. `sys.columns`
- Compatibility views – backward compatible with older tables e.g. `syscolumns`

Where possible you should use catalog views, which like DMVs, are part of the `sys` schema. Catalog views contain both server and database level objects, and tend to be more user-friendly (for example, better named columns) than the older system tables.

1.5.2 Cached plans

When a query is run, a cached plan for it is created. This details what tables are accessed, what indexes are used, and what types of joins are performed etc. Storing this information in a cached plan allows subsequent similar queries (where the parameters differ) to re-use this plan, thus saving time.

UNDERSTANDING THE CACHED PLAN

In many ways, a cached plan is analogous to the well-trodden tourist excursion many of us have undertaken when we take a holiday. The experienced tour guide knows the most efficient routes to use to fulfill the experiences of his group of tourists. Similarly, the SQL Server optimizer knows the most efficient routes to access different tables for its SQL queries.

When using the DMVs that relate to SQL queries, we will often look at the query's cached plan to get a greater insight into how the query is fulfilling its requirements (for example, getting or updating data). This will allow us to target many problems and get ideas of any possible improvements.

Examining a query's cached plan can give you a great deal of insight into why a query is running slowly. Maybe the query is not using an index? Maybe the query is using the 'wrong' index? Maybe the data's statistics are out of date? All this information and more can be gleaned by examining the cached plan.

Our output from our sample SQL snippets will often contain the cached plan associated with the SQL, understanding it will give insight into how query currently works and how you might want to change it to

improve its performance. For example, add a missing index or update any stale statistics. We'll hear more about reading cached plans later.

Luckily, we can use DMVs to access the cached plans, allowing us to investigate further why the query is having problems and potentially provide solutions. We saw earlier, in the section entitled "Quickly find a cached plan" how a cached plan can be used to target the area costing the most in terms of performance.

1.5.3 Indexes

Perhaps the main tool for improving the performance of your queries is the index. Indexes are used for fast retrieval of data (imagine looking for something specific in this book without the index at the back!). Additionally, indexes are also useful for sorting and providing both unique and foreign key constraints.

The DMVs record many index-related details including how often an index is used, how it is used (as part of a scan, lookup, by the application or system routines), whether it is not used at all, any concurrency problems accessing the indexes, and indeed details of any missing indexes.

Knowing about how the different types of indexes are used will give you greater pool of knowledge from which you can propose solutions. In essence, for retrieving a small number of relatively unique rows you want an index that can quickly identify the subset of rows. These are typically non-clustered indexes. For longer reporting-like queries you typically want a range of rows that are hopefully physically next to each other (so you can get them with fewer reads). This typically means a clustered index. We will discuss indexes in more detail in chapter 3 relating to index DMVs.

1.5.4 Statistics

When a query is run, the optimizer inspects the relevant tables, row counts, constraints, indexes and data statistics to determine a cost effective way of fulfilling the query. Statistics describe the range and density of the data values for a given column. This is used to help determine the optimal path to the data. This information is used to create a plan that is cached for re-use. In essence, statistics can greatly influence how the underlying data is queried.

When the data in the table changes the statistics may become stale, this may result in a less efficient plan for the available data, to get around this, the statistics are typically updated automatically, and any necessarily plans recompiled and re-cached.

For tables with more than 500 rows, it requires a 20% change in the underlying data before the statistics are automatically updated. For large tables, containing, for example 10 million rows, it would require 2,000,000 changes before the statistics are recalculated automatically. If you were to add 100,000 rows to this table on a daily basis, it would require 20 days before the statistics are updated, until that time you may be working with stale statistics and a suboptimal plan. Knowing this, it is often advisable to update the statistics more regularly using a scheduled job. I have experienced many occasions when queries have been running very slowly, run almost instantaneously when the table's statistics are updated.

In many ways, especially for larger tables, on mature systems, I feel statistics are a critical element in the efficiency of database systems. When you run a query, the optimizer looks at the columns you join on, together with the columns involved with your WHERE clause. It looks at the column data's statistics to determine the probabilities involved in retrieving data based on those column values. It then uses these statistical probabilities to determine if an index should be used, and how it should be used (for example, seek, lookup or scan). So you can see, having up to date statistics is very important, later in the book (chapter 3, section 3.10 entitled "Your Statistics") I'll show you a SQL script to determine if your statistics need to be refreshed.

1.6 Working with DMVs

Problems can be tackled from several angles, using a variety of tools. The point to note is some tools are more appropriate than others for given tasks. For example, you could use a screwdriver or a blunt knife to undo a screw, both could probably do the job, but you'll find one is easier than the other. Similarly, if you want to determine which queries are running slowly, you could use SQL Server Profiler. However, a quicker/smarter way would be to use DMVs.

This is not to say that using DMVs are better than other tools. The point I want to make is that sometimes, depending on the problem you are trying to investigate, using DMVs may provide a quicker and easier method of investigation. The different tools should be seen as complimentary, rather than mutually exclusive.

Part of the problem of using DMVs is that they tend to be little known and untried compared with the more established tools. Hopefully the code samples given in this book will help form the basis of an additional approach to problem solving.

1.6.1 In context with other tools

Developers and DBAs that lack knowledge of DMVs will typically turn to the traditional problem solving database tools including tracing, cached plan inspection, database tuning advisor, and performance monitor. These are discussed briefly below in the comparison with using DMVs.

SQL SERVER PROFILER

SQL Server comes with a SQL Server Profiler utility that allows you to record what SQL is running on your SQL Server boxes. It is possible to record a wide range of information (for example, number of reads/writes or query duration) and filter the range of data you want to record (for example, for a given database or spid).

SQL Server Profiler is a well known and much used utility, typically allowing you to target the cause of a problem. However it does use system resources and because of this running it on production systems is usually not recommended.

There are various reasons for using SQL Server Profiler, including discovering what SQL queries are being run, why a query is taking so long to run, and creating a suite of SQL that can be replayed later for regression testing. We have already seen in the DMV examples section how we can discover both what is running and also the slowest queries easily and simply by using the DMVs. With this in mind, it may be questionable if we need to use SQL Server Profiler to capture information that is already caught by the DMVs (remember we can get the delta between two DMV snapshots to determine the effect of a given batch of SQL queries).

Looking further at using SQL Server Profiler to discover why a batch of SQL is running slowly, we have the additional task of summing the results of the queries, some of which may run quickly but are run often (so their accumulative effect is large). This problem is compounded by the fact that the same SQL may be called many times but with different parameters. Creating a utility to sum these queries can be time consuming. This summation is done automatically with the DMVs.

In chapter 11 (Useful scripts) section 11.12, I will present a simple and lightweight DMV alternative to the SQL Server Profiler.

DATABASE TUNING ADVISOR

The Database Tuning Advisor (DTA) is a great tool for evaluating your index requirements. It takes a given batch of SQL statements as its input (for example, taken from a SQL Server Profiler trace), and based on this input, it determines what the optimal set of indexes are to fulfill those queries.

The SQL statements used as input into the DTA should be representative of the input you typically process. This should include any special processing, such as month end or quarterly processing. It can also be used to tune a given query precisely to your processing needs.

The DTA amalgamates the sum total effect of the SQL batch and determines if the indexes are worthwhile. In essence, it evaluates if the cost of having a given index for retrieval is better than the drawbacks of having to update the index when data modifications are made.

Where possible, the indexes the DTA would like to add or remove should be correlated with those proposed by the DMVs, for example, missing indexes, unused or high-maintenance indexes. This shows how the different tools can be used to compliment each other rather than being mutually exclusive.

PERFORMANCE MONITOR

Performance monitor is a Windows tool that can be used to measure SQL Server performance via various counters. These counters relate to objects such as processors, memory, cache, and threads. Each object in turn has various counters associated with it to measure such things as usage, delays, and queue

lengths. These counters can be very useful in determining if a resource bottleneck exists, and where further investigation should be targeted.

In SQL Server 2008 it is possible to merge the Performance Monitor trace into the SQL Server Profiler trace, enabling us to discover what is happening in the wider world of Windows when given queries are run.

These counters measure various components that run on Windows. A subset of them, that relate to SQL Server in particular, can be accessed via the DMV `sys.dm_os_performance_counters`, we will discuss these in chapter 6 (Operating System DMVs). If we query this DMV at regular intervals and store the results, we can use this information in diagnosing various hardware and software problems.

CACHED PLAN INSPECTION

We've already discussed how we can get the cached plan for a given query, and also its importance in relation to DMVs. Having a cached plan is a great starting point for diagnosing problems, since they provide more granular details of how the query is executed.

Each SQL statement within a batch is assigned a percentage cost in relation to the whole of the batch. This allows you to quickly target the query taking most of the query cost. For each query the cached plan contains details of how that individual query is executed, for example, what indexes are used, and the index access method. Again a percentage is applied to each component. This allows you to quickly discover the troublesome area within a query that is the bottleneck.

In addition to investigating the area identified as being the mostly costly, we can also check the cached plan for indicators of potential performance problems. These indicators include table scans, missing indexes, columns without statistics, implicit data type conversions, unnecessary sorting, and unnecessary complexity. We will provide a SQL query later in the book that will allow you to search for these items that may be the cause of poor performance.

DMVs are typically easier to extract results from, when compared with other more traditional methods. However, these different methods are not mutually exclusive, and where possible, the different methods should be combined to give greater support and insight into the problem being investigated.

1.6.2 Self-healing database

Typically we get notified of a problem via an irate user, for example, "my query's taking too long, what's happening?" Whilst identifying and solving the problem using a reactive approach fixes the immediate difficulty, a better, more stress free and professional approach would be to be preemptive, to identify and prevent problems before they become noticeable.

A preemptive approach involves monitoring the state of your databases, and automatically fixing any potential problems before they have a noticeable effect. Ultimately, if we can automatically fix enough of these potential problems before they occur, we will have a self-healing database.

If we adopt a preemptive approach to problems, with a view to fixing potential problems before they become painfully apparent, we can implement a suite of SQL Server jobs that run periodically, that can not only report potential problems but also attempt to fix them too. With the spread and growth of SQL Server within the enterprise via tools such as SharePoint, and Customer Relationship Management (CRM) systems, as well as various ad-hoc developments (that have typically been outside the realms of database developers or DBAs), there should be an increasing need for self-healing databases, and a corresponding increase in knowledge of DMVs.

If we take as our goal the premise that we want our queries to run as quickly as possible, then we should be able to identify and fix issues that counteract this aim. Such issues include missing indexes, stale statistics, index fragmentation, and inconsistent data types.

I would hope we can provide SQL queries that run as regular SQL Server jobs that will at least attempt to automate the fixing of these issues with a view creating a self-healing database. These queries will report on the self-healing changes and if necessary implement the self-healing changes. These queries will be provided in chapter 10 entitled "The self-healing database".

1.6.3 Reporting and transactional databases

Using DMVs you could present a case for separating out the reporting aspects of the database from the transactional aspects. This is important because they have different uses and they result in different optimal database structures. Having them together often produces conflicts.

A reporting database is one primarily concerned with retrieving data. Some aggregation may have already been done else is done dynamically, as it is required. The emphasis is on reading and processing lots of data, often resulting in a few, but long running, queries. To optimize for this we tend to have lots of indexes (and associated statistics), with a high degree of page-fullness (so we can access more row per read). Typically, data doesn't have to appear in the reporting database 'immediately'. Often we are reporting on how yesterday's data compares with previous data, so potentially it can be up to 24 hours late. Additionally, reporting databases have more data (indexes are often very large), resulting in greater storage, longer backups and restores.

By comparison, a transactional database is one where the queries typically retrieve and update a small number of rows, and run relatively quickly. To optimize for this, we tend to have few indexes, with a medium degree of page fullness (so we can insert data in the correct place without causes too much fragmentation).

Now we've outlined the differing needs of both the reporting and transactional databases I think you can see how their needs compete and interfere with each other's optimal design. If our database has both reporting and transactional requirements, then when we update a row, if there are additional indexes these too will need to be updated, resulting in a transactional query that takes longer to run, leading to a greater risk of blocking, timeout (in .NET clients for example), and deadlock. Additionally the transactional query, although it would run quickly, might be blocked from running by a long running reporting query.

We can look at the DMVs to give us information about the split of reporting versus transactional requirements, this data includes:

- number of reads versus the number of writes per database or table
- number of missing indexes
- number and duration of long running queries
- number and duration of blocked queries
- space taken (also reflects time for backup/restore)

Usually, the missing indexes need to be treated with caution. Whilst adding indexes is great for data selection (reporting database), it may be detrimental to updates (transactional database). If the databases are separated, we can easily implement these extra indexes.

With a reporting database will can create the indexes such that there is no redundant space on the pages, thus ensuring we optimize data retrieval per database read. Additionally, we could mark the database (or specific file-groups) as read-only, thus eliminating the need for locking and providing a further increase in performance

Using this data can help determine if separating out at least some of the tables into another database might lead to a better database strategy (for example, tables that require lots of IO could be placed on different drives, allowing improved concurrent access). There are many ways of separating out the data including replication, and mirroring.

1.7 Summary

This chapter's short introduction to Dynamic Management Views has illustrated the range and depth of information that is available quickly, easily and freely, just for the asking.

We've discovered what DMVs are, and the type of problems they can solve. DMVs are primarily used for diagnosing problems and also assist in the proposal of potential solutions to these problems.

Various example SQL snippets have been provided and discussed. These should prove immediately useful in determining your slowest SQL queries, identifying your mostly costly missing indexes, identifying what SQL statements are running on your server now, and retrieving the cached plan for an already executed query. Additionally, a very useful simple monitor has been provided.

The rest of the book will provide many useful example code snippets, which cover specific categories of DMVs, but always with a focus on the developer's/DBA's needs. Since we tend to use similar patterns for many of the SQL snippets, it makes sense to discuss these common patterns first, which I will do in the next chapter.