SAMPLE CHAPTER

SQL Server DMVs IN ACTION

Better queries with Dynamic Management Views

Ian W. Stirk

MANNING

SQL Server DMVs in Action

by Ian W. Stirk

Chapter 1

brief contents

1 The Dynamic Management Views gold mine

This chapter covers

- What Dynamic Management Views are
- Why they're 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 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, and they 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 in the next section.

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

I'm sure that after reading this chapter you'll be pleasantly surprised when you discover the wealth of information that's available for free within SQL Server that can be accessed via DMVs and the impressive impact using this information can have. The DMV data is already out there waiting to be harvested; in so many ways it's a gold mine!

1.1 What are Dynamic Management Views?

As queries run on a SQL Server database, SQL Server automatically records information about the activity that's taking place, internally into structures in memory; you can access this information via DMVs. DMVs are basically SQL views on some pretty important internal memory structures.

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

DMV information is stored on a per–SQL Server instance level. You can, however, provide filtering to extract DMV data at varying levels of granularity, including for a given database, table, or 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. In addition, many corollary areas enhance and extend the DMV output. I'll discuss these a little later, in the section titled "DMV companions."

Don't worry if you're not familiar with all these terms; the purpose of this book is to help explain them and present examples of how you can use them to improve the performance and your 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. This book takes a similar approach. A brief outline of each of the DMV categories follows in table 1.1.

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 group of DMVs occurs in SQL Server 2008 and higher.
Common Language Runtime	The Common Language Runtime allows code that runs on the database 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 and smaller than a session), loaded assemblies, properties, and running tasks.
Database	These DMVs contain information relating to various aspects of databases, including space usage, partition statistics, and session and task space information.
Database mirroring	The aim of database mirroring is to increase database availability. Transaction logs are moved quickly between servers, allowing fast failover to the standby server. These DMVs contain information relating to various aspects of database mirroring, including connection information and page-repair details.
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.
Extended events	Extended events allow SQL Server to integrate into Microsoft's wider event-handling processes, allowing integration of SQL Server events with logging and monitoring tools. This group of DMVs occurs in SQL Server 2008 and higher.
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 catalogs, 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, and 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).

Table 1.1 The major DMV groups *(continued)*

DMV group	Description
Input/Output (I/O)	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 I/O stall time), backup tape devices, and any pending I/O 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), transactions, 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 that 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. This group of DMVs occurs in SQL Server 2008 and higher.
Service Broker	Service Broker is concerned with providing both transactional and 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 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).

Because this book takes a look at DMVs from a practical, everyday troubleshooting and maintenance perspective, it concentrates on those DMVs that the DBA and database developer will use to help solve their everyday problems. With this in mind, it concentrates on the following categories of DMV:

- Index
- Execution
- SQL Server Operating System

- Common Language Runtime
- Transaction
- Input/Output
- Database

If there's sufficient subsequent interest, perhaps another book could be written about the other DMV 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 following:

- 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 can't, because they're missing
- How much I/O occurred (both physical and logical)
- How much time was spent executing the query
- How much time was spent waiting on other resources
- What resources the query was waiting on

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, several related functions 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 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, run the following:

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

Note that 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'll 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 cumulative since the last SQL Server reboot or restart. Often this is useful, because you want to know the sum total effect for each of the queries that have run on the server instance or a given database.

But if you're interested only in the actions of a given run of a query or batch, you can determine the effect of the query by taking a snapshot of the relevant DMV data, run your query, and then take another snapshot of the DMV data. Getting the delta between the two snapshots will provide you 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, "Calculating DMV changes."

1.1.3 Impact of running DMVs

Typically, when you query the DMVs to extract important diagnostic information, this querying has a minimal effect on the server and its resources. This is because the data is in memory and already calculated; you 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 I'll highlight them 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 onward

DMVs and DMFs have been an integral part of SQL Server since version 2005. 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 2005 and higher. It's 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 2005, getting the level of detailed information given by DMVs is difficult or impossible. For example, to obtain details of the slowest queries, you'd typically have to run SQL Trace (this is the precursor of SQL Server Profiler) for a given duration and then spend a considerable amount of time analyzing and aggregating the results. This was made more difficult because the parameters for the same queries would often differ. The corresponding work using DMVs can usually be done in seconds.

1.2 The problems DMVs can solve

In the section titled "What are Dynamic Management Views?" I briefly mentioned the different types of data that DMVs record. I can assure you that this range is matched by depth too. DMVs allow you to view a great deal of internal SQL Server information that's 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.

> **NOTE** DMVs aren't the sole method of targeting the source of a problem or improving subsequent performance, but they can be used with other tools to identify and correct concerns.

The problems DMVs can solve can be grouped into diagnosing, performance tuning, and monitoring. In the following sections I'll discuss each of these in turn.

1.2.1 Diagnosing problems

Diagnosing problems is concerned with identifying the underlying cause of a problem. This is perhaps the most common use of DMVs. It's possible to query the DMVs to diagnose many common problems, including your slowest queries, the most common causes of waiting/blocking, unused indexes, files having the most I/O, and lowest reuse of cached plans. Each of these areas of concern and more could be a starting point 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's 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 you to use the DMVs at each of these levels.

Sometimes, identified problems aren't real problems. For example, there may be queries that run slowly but they run at a time when it doesn't cause anyone any concern. So although you could fix them, it would be more appropriate to focus your problem-solving skills on issues that are deemed 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, you 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), or if the statistics are out of date, as well as to identify any missing indexes and to target the particular statement or access path that's causing the slowness. Later we'll 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 effect on subsequent performance. Applying these new features leads us into the area of performance tuning. We'll 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, you should try breaking it down into smaller steps. Not only might this highlight the problem area with finer granularity, but it might also solve it! Maybe the optimizer has more choices available to it with simpler queries and generates a better plan. You can see if this is the case by examining the relevant execution DMVs, as will become clear in chapter 5.

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, determining the degree of fragmentation, and so on. Again, the query's cached plan is a primary source of ideas for improvement.

Measurement of any improvement is typically reflected in time or I/O counts and can be made with traditional tools such as turning on STATISTICS IO or STATISTICS TIME SQL commands or using a simple stopwatch. But for more comprehensive results, you 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 two times indicates a high degree of waiting/blocking may be occurring. Similarly, DMVs also record the amount of I/O (reads/writes at both the physical and logical level) that can be used to measure the effectiveness of a query, because less I/O typically reflects a faster query.

Again, you 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 you need to ask if any remaining problem is worth solving, because you should always aim to fix what is deemed to be the most important problems first.

You need to be careful of the impact performance-based changes can have on the maintainability of systems; often these two needs are diametrically opposed because complexity is often increased. Rather than guess where optimization is needed, you should undertake appropriate testing first to determine where it's 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."[1] I'll discuss this in more detail in chapter 5.

1.2.3 Monitoring

A large group of DMVs (those starting with sys.dm_exec_) relates to what's currently executing on the server. By repeatedly querying the relevant DMVs, you get a view of

[1] Donald Knuth, "Structured Programming with go to Statements," *ACM Journal Computing Surveys* 6, no. 4 (December 1974): 268.

the status of the server instance and also its history. Often this transient information is lost, but it's possible to store it for later analysis (for example, into temporary or semi-permanent tables). An example of this is given in chapter 11, section 11.7, titled "Who's doing what and when?"

Sometimes you have problems with the overnight batch process, reported as a 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.

Although 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? You can get this information by using DMVs combined with a simple monitoring script. I've used such a script often to examine problems that occur during an overnight batch run.

> **NOTE** This example uses routines I've created and fully documented in the web links given in the following code sample (so you see, not only is code reuse good but article reuse 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: mng.bz/V5E3; and look here for the routine named dba_WhatSQLIsExecuting: mng.bz/uVs3). 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 two utilities with your own favorite utilities. Later in this chapter I'll go through the code that forms the basis of one of the stored procedures (dba_WhatSQLIsExecuting).

The following listing shows the code for a simple monitor.

Listing 1.1 A simple monitor

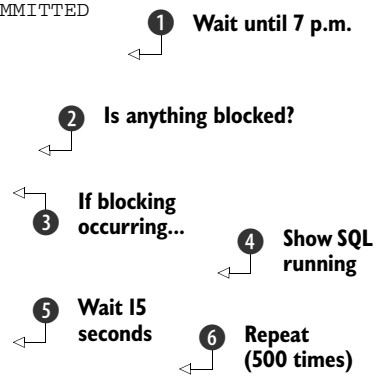
```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

WAITFOR TIME '19:00:00'
GO

PRINT GETDATE()
EXEC master.dbo.dba_BlockTracer

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

WAITFOR DELAY '00:00:15'
GO 500
```



This code snippet waits until a specified time (7 p.m. in this example ❶) and then prints the date/time and runs a routine named dbo.dba_BlockTracer ❷. 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 nonzero value ❸. This causes it to output the date and time and list all the SQL that's running ❹ (including the batch/stored procedure and the individual SQL statement within it that's running). The utility then waits a specified time (15 seconds in this example ❺) and repeats. All this is repeated (except waiting until 7 p.m.) a number of times, as specified by the last GO statement (500 in this example ❻).

The routines show not only what's 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'll notice that the output in the various grids contain spids that can be used to link the output from the two utilities together. An example of the type of output for this query is given in figure 1.1.

The first two 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's currently running are shown; these include the individual line of SQL that's 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 you enter a similar batch of SQL statements into two or more distinct windows within SQL Server Management Studio (SSMS), and the statements are followed with a GO 5000 and the SQL in all windows run at the same time, you can discover the effect of repeatedly running the SQL at the same time.

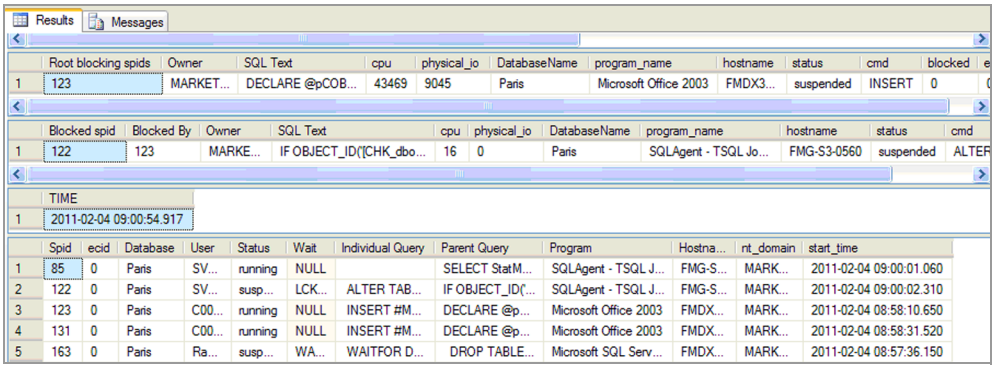
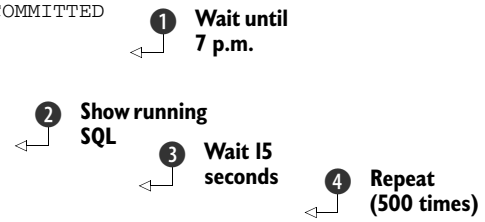


Figure 1.1 Output showing if anything is blocked and what individual SQL queries are running

It's possible to determine what's running irrespective of any blocking by using an even simpler monitoring query, given in the following snippet:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
WAITFOR TIME '19:00:00'
GO
SELECT GETDATE() AS TIME
EXEC master.dbo.dba_WhatSQLIsExecuting
WAITFOR DELAY '00:00:15'
GO 500
```



The query waits for a given time (7 p.m. ❶) and then displays the date and time together with details of what SQL queries are running ❷. It then waits for a specified period (15 seconds ❸) and repeats ❹ (but doesn't wait until 7 p.m. 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. You can query the DMVs to determine what queries are waiting (being blocked) the most and aim to improve them. We'll identify the most-blocked queries later, in chapter 4, section 4.5, "Queries that spend a long time being blocked."

You can use the simple monitor utility discussed previously to determine why these identified queries are being blocked; the DMVs will tell you what is blocked, but they don't identify what's blocking them. The monitoring utility can do this. The monitor utility can be a 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 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 won't 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 you can read data without waiting for locks to be released or acquiring locks yourself, 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's often the case that you have several different databases running on the same server. A consequence of this is that 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's 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, because queries are often reported as running too slowly. If you run the SQL query given in the following listing, you'll 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(qs.total_elapsed_time / 1000000.0 AS DECIMAL(28, 2))
                                    AS [Total Elapsed Duration (s)]
  , qs.execution_count
  , SUBSTRING (qt.text,(qs.statement_start_offset/2) + 1,
    ((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) qt
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY total_elapsed_time DESC
```

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 first CROSS APPLY takes a parameter (sql_handle) and retrieves the text of the query. The second CROSS APPLY takes another parameter (plan_handle) and retrieves

Results | Messages

	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 #AllPNL 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 cached plan associated with the query The cached plan is a primary resource for discovering why the query is running slowly, and often it 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 ❸. To limit the amount of output, only the slowest 20 queries are reported. Running the slowest-queries query on my server gives the results shown in figure 1.2.

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's 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 section 2.10, "Calculating DMV changes."

The NULL values in the DatabaseName column mean the query was run either ad hoc or using prepared SQL (that is, not as a stored procedure). This in itself can be interesting because it indicates areas where stored procedures aren't being reused 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. But for various reasons, for example, inexperienced developers or changing systems, useful indexes may not always have been created. Running the SQL query given in the next listing 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(s.avg_total_user_cost * s.avg_user_impact *
     (s.user_seeks + s.user_scans),0) AS [Total Cost]
  , s.avg_user_impact
  , d.statement AS TableName
  , d.equality_columns
  , d.inequality_columns
  , d.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
```

❶ Calculate cost

❷ Sort by cost

The DMV sys.dm_db_missing_index_group_stats contains metrics for missing indexes, including 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 two DMVs (metrics and names) are linked together via another DMV, sys.dm_db_missing_index_groups, which returns information about missing indexes in a specific missing index group.

You should note how the Total Cost field of the missing index is calculated ❶. 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 ❷.

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 displays the results shown in figure 1.3.

Results | Messages

	Total Cost	avg_user_impact	TableName	EqualityUsage	InequalityUsage	Include Clouns
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	[PNLAdjustmentQueueI...
5	282331	20.41	[ParisDev].[dbo].[PNLAdjustme...	NULL	[Status]	[PNLAdjustmentQueueI...
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]	[PNLAdjustmentQueueI...
10	73727	12.98	[ParisDev].[dbo].[PNLAdjustme...	NULL	[LocalValue], [...	[PNLAdjustmentQueueI...

Figure 1.3 Output from the 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, I'll discuss many aspects of index usage throughout this book, and especially in chapter 3, "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 gotten 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's being executed (this can be used to monitor progress of a batch of SQL). Note that 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'll see that we ignore any system processes (having a spid of 50 or less), and we also ignore this actual script.

Listing 1.4 Identify what SQL is running now

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
  er.session_Id AS [Spid]
  , sp.ecid
  , DB_NAME(sp.dbid) AS [Database]
  , sp.nt_username
  , er.status
  , er.wait_type
  , SUBSTRING (qt.text, (er.statement_start_offset/2) + 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]
  , sp.program_name
  , sp.Hostname
  , sp.nt_domain
  , er.start_time
```

1 **Extract SQL statement**

```
FROM sys.dm_exec_requests er
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
```

❷ Join request to sysprocesses

The DMV sys.dm_exec_requests contains details of each request, the SQL query ❶, executing on SQL Server. This DMV is joined to the catalog view sys.sysprocesses ❷ based on its session id. Catalog views are similar to DMVs but contain static data; I will talk more about them shortly, in the section "DMV companions." The catalog view sys.sysprocesses contains information about the environment from which the request originated and includes such details as user name and the name of the host it's running from. Combining the DMV and catalog view gives you 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's currently running. Running the "what SQL is running now" query on my server gives the results shown in figure 1.4.

Results | Messages

	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 identifies which 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 (whether the SQL is running or waiting), the wait status (why it's waiting), the hostname, the domain name, and the start time (useful for determining how long the batch has been running). I'll explain these columns and their relevance in detail later in the book, in chapter 5, section 5.9, "Current running queries."

You can see the route a SQL query takes in answering a query 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 cached 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's 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 reusing this plan.

It's possible to obtain the estimated or actual execution plan for a batch of SQL by clicking the relevant icon in SQL Server Management Studio. Typically the estimated

plan differs from the actual plan in that the former isn't 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).

But there are problems with this approach. It may not be viable to run the query because 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' ❶, of which there should be many. Note that you'll need to enter some text that uniquely identifies your SQL, for example, the stored procedure name, to retrieve the specific cached plans you'd 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
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%'
```

Join cached plan and SQL text DMVs

❶ **Text to search plan for**

Running the query from listing 1.5 on my server gives the results shown in figure 1.5.

Results | Messages

	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

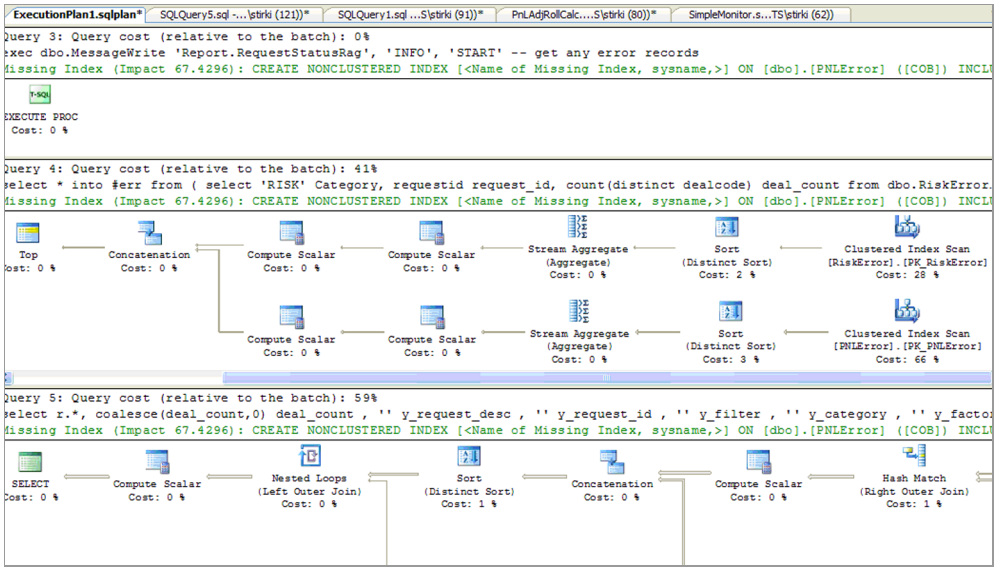


Figure 1.6 Missing indexes details included with a 2008 cached plan

When you identify the relevant query you want the cached plan for (the previous query is quite generic, looking for plans that contain the text 'CREATE PROCEDURE'), clicking the relevant row under the column named query_plan will display the query plan. How it does this differs depending on whether you're using SQL Server version 2005 or 2008. If you're using version 2005, clicking the column will open 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 (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 the query_plan column will open the cached plan as a full graphical version; this is shown in figure 1.6.

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'll be given at the top of each section in green, with text starting "Missing Index" (again see figure 1.6). If you right-click the diagram, you can select Missing Index Details. Clicking this will open a new window with a definition of the missing index ready to add; you just need to add an appropriate index name. An example of this is shown here.

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 SwapsDiaryFile, I can quickly get its cached plan, part of which is shown in figure 1.7.

Looking at figure 1.7, you can see that each statement within a batch or stored procedure has a query cost associated with it (here you'll notice that the first two 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 three parts, with cost values of 0%, 62%, and 38%. You can

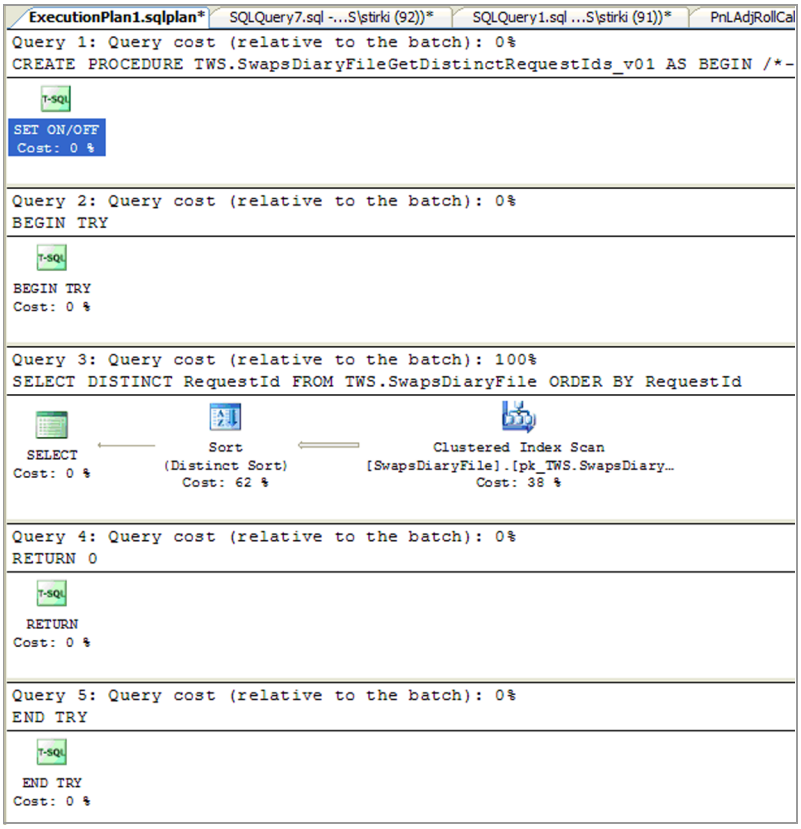


Figure 1.7 Cached plan showing cost by statement and within each statement

thus identify the section within the batch that should be targeted for improvement; in our case it's the operation that cost 62%.

Before DMVs can be used, you need to ensure that the appropriate permissions have been assigned and that you're aware of the accumulative nature of DMVs; we discuss these next.

1.4 Preparing to use DMVs

A great many exciting DMV insights into our SQL queries await us. But before you can dive into using DMVs, you need to ensure that you're 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 DMV and DMF usage, namely, *server-scoped* and *database-scoped*. Server-scoped requires VIEW SERVER STATE permission on the server, and database-scoped requires VIEW 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.

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

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

1.4.2 Clearing DMVs

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

Before we discuss how the DMVs can be cleared, it's worth noting that another approach exists to allow you to determine the effect of a given query or system action. DMV data is cumulative, so to determine the effect of a query, you can to take a snapshot of the relevant DMVs, run the query under investigation, and then take a second

Clearing DMVs

Please note that you should clear DMVs on production machines only after careful consideration, because this will result in queries taking longer to run because they need to be compiled again.

snapshot. You can determine the effect of the query 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 titled "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), because 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. While this may be the easiest method, it's probably also the most drastic in terms of impact on users, so you should use it with caution.

Alternatively, it's 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 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, on SQL Server 2008 and higher, is either a plan_handle, sql_handle, or pool_name. The plan_handle and sql_handle are 64-bit 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.

You can also clear the cached plans for a specific database only, using the following commands:

```
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 methods of creating a more permanent version of this transient information for later analysis. An example of this is given in section 11.7, "Who is doing what, and when?"

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

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

1.5 DMV companions

Although this book is primarily concerned with the usage of DMVs, to get the most out of the DMVs it's 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 you a better understanding of what the DMVs provide and what you can do 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, you'll need to join the DMVs with various catalog views to produce more meaningful output. For example, when you're processing DMVs that relate to indexes, you'll 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 metadata in system tables; these tables are still present in later versions. But it's not recommended to query these system tables directly, because future internal changes that Microsoft makes may break your code. These system tables have been replaced by the following:

- *Catalog views*—Tables that describe objects, for example, sys.columns
- *Compatibility views*—Backward compatible with older tables, for example, 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, what types of joins are performed, and so on. Storing this information in a cached plan allows subsequent similar queries (where the parameters differ) to reuse this plan, saving time.

When using the DMVs that relate to SQL queries, you'll 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 you 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 isn't using an index. Maybe the query is

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 vacation. The experienced tour guide knows the most efficient routes to use to fulfill the expectations of the group of tourists. Similarly, the SQL Server optimizer knows the most efficient routes to access different tables for its SQL queries.

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.

The output from your sample SQL snippets will often contain the cached plan associated with the SQL; understanding it will give insight into how a query currently works and how you might want to change it to improve its performance, for example, adding a missing index or updating any stale statistics. You'll hear more about reading cached plans later.

Luckily, you can use DMVs to access the cached plans, allowing you to investigate further why the query is having problems and potentially provide solutions. You saw earlier, in the section titled "Quickly find a cached plan," how you can use a cached plan 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 unique constraints.

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

Knowing about how the different types of indexes are used will give you a 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 nonclustered 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'll discuss indexes in more detail in chapter 3.

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. These are used to help determine the optimal path to the data. This information is used to create a plan that's cached for reuse. In essence, statistics can greatly influence how the underlying data is queried.

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

For tables with more than 500 rows, a 20% change in the underlying data is required before the statistics are automatically updated. For large tables, containing,

for example, 10 million rows, 2 million changes would be necessary 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. Because of this, it's often advisable to update the statistics more regularly using a scheduled job. I've experienced many occasions when queries have been running slowly, but they 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 whether an index should be used and how it should be used (for example, seek, lookup, or scan). As you can see, having up-to-date statistics is important. Later in the book (chapter 3, section 3.10, "Your statistics"), I'll show you a SQL script to determine whether your statistics need to be refreshed.

1.6 Working with DMVs

You can tackle problems 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, but a quicker, smarter way would be to use DMVs.

This isn't to say that DMVs are better than other tools. The point I want to make is that sometimes, depending on the problem you're trying to investigate, using DMVs may provide a quicker and easier method of investigation. The different tools should be seen as complementary 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 who lack knowledge of DMVs typically turn to the traditional problem-solving database tools, including tracing, cached plan inspection, Database Tuning Advisor, and Performance Monitor. These are discussed briefly in 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's 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. But 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, determining why a query is taking so long to run, and creating a suite of SQL that can be replayed later for regression testing. You've already seen in the DMV examples section how you can discover both what is running and the slowest queries easily and simply by using the DMVs. With this in mind, it may be questionable whether you need to use SQL Server Profiler to capture information that's already caught by the DMVs (remember that you 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, you 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, section 11.12, I'll present a simple and lightweight DMV alternative to 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 the optimal set of indexes 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. The DTA 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 whether the indexes are worthwhile. In essence, it evaluates whether 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, you should correlate the indexes the DTA would like to add or remove with those proposed by the DMVs, for example, missing indexes or unused or high-maintenance indexes. This shows how the different tools can be used to complement 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

useful in determining whether a resource bottleneck exists and where further investigation should be targeted.

In SQL Server 2008 it's possible to merge the Performance Monitor trace into the SQL Server Profiler trace, enabling you to discover what's 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 relates to SQL Server in particular can be accessed via the DMV sys.dm_os_performance_counters; I'll discuss these in chapter 6 ("Operating system DMVs"). If you query this DMV at regular intervals and store the results, you can use this information in diagnosing various hardware and software problems.

CACHED PLAN INSPECTION

We've already discussed how you 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, because it can 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 that's 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's the bottleneck.

In addition to investigating the area identified as being the mostly costly, you 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'll 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. But these different methods aren't mutually exclusive, and where possible, you should combine the different methods 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?" Although identifying and solving the problem using a reactive approach fixes the immediate difficulty, a better, more stress-free and professional approach would 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 you can automatically fix enough of these potential problems before they occur, you'll have a self-healing database.

If you adopt a preemptive approach to problems, with a view to fixing potential problems before they become painfully apparent, you can implement a suite of SQL Server jobs that run periodically, which can not only report potential problems but also attempt to fix them. 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 you take as your goal the premise that you want your queries to run as quickly as possible, then you 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.

Later in this book I'll 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 toward 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, "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 or else is done dynamically as it's 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 rows per read). Typically, data doesn't have to appear in the reporting database immediately. Often we're 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 and 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 causing too much fragmentation).

Now that I'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 your database has both reporting and transactional requirements, then when you 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.

You can look at the DMVs to give you information about the split of reporting versus transactional requirements. This data includes the following:

- 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. Although adding indexes is great for data selection (reporting database), it may be detrimental to updates (transactional database). If the databases are separated, you can easily implement these extra indexes on the reporting database.

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

Using this data can help you determine whether 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 I/O 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's available quickly, easily, and freely, just for the asking.

You'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. In addition, a 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. Because we tend to use similar patterns for many of the SQL snippets, it makes sense to discuss these common patterns first, which I'll do in the next chapter.

SQL SERVER ADMINISTRATION

SQL Server DMVs IN ACTION

Ian W. Stirk

Free ebook
SEE INSERT

Every action in SQL Server leaves a set of tiny footprints. SQL Server records that valuable data and makes it visible through Dynamic Management Views, or DMVs. You can use this incredibly detailed information to significantly improve the performance of your queries and better understand what's going on inside your SQL Server system.

SQL Server DMVs in Action shows you how to obtain, interpret, and act on the information captured by DMVs to keep your system in top shape. The over 100 code examples help you master DMVs and give you an instantly reusable SQL library. You'll also learn to use Dynamic Management Functions (DMFs), which provide further details that enable you to improve your system's performance and health.

What's Inside

- Many practical solutions
- How to correct missing indexes
- What's slowing down your queries
- What's compromising concurrency
- Much more

This book is written for DBAs and developers.

Ian Stirk is a freelance consultant based in London. He's an expert in SQL Server performance and a fierce advocate for DMVs.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/SQLServerDMVsinAction

"Essential reference for SQL Server Administrators."
—Dave Corun, Avanade

"Arm yourself with an arsenal of DMV knowledge."
—Tariq Ahmed
Amcom Technology

"Lifts the hood on SQL Server performance."
—Richard Siddaway, Serco

"The examples alone are worth *twice* the price of the book!"
—Nikander and Margriet Bruggeman
Lois & Clark IT Services

ISBN 13: 978-1-935182-73-3
ISBN 10: 1-935182-73-0
56499
9 781935 182733

MANNING $64.99 / Can $74.99 [INCLUDING eBOOK]