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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
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). DMV queries 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 green paper you’ll learn what DMVs are, the kinds of data they contain, and the types of problem they can solve. We’ll provide several example code snippets that you’ll find immediately useful. We’ll briefly discuss DMVs in the context of other problem-solving tools and related structures (for example, indexes and statistics). Finally, we’ll outline the major groups the DMVs are divided into and the ones we’ll be concentrating on.

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 goldmine!

**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; this information can be accessed via DMVs. DMVs are basically SQL views on some important internal memory structures.

Lots of different types of information are recorded, which 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 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.

**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)
- Which indexes were used
- Which indexes the query would like to use but are missing

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
• 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
• Which resources the query is 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, called 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 called `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; when you reference them you must supply this schema name. For example, to query the `dm_exec_requests` DMV, 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.

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

But if we’re interested only in the actions of a given run of a query or batch, we can determine the affect 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.

**Impact of running DMVs**
Typically, when we 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, and 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`).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
In summary, compared with other methods of obtaining similar information, for example, by using the Database Tuning Advisor or SQL Trace, using DMVs is relatively unobtrusive and has little impact on the system performance.

**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. It’s possible to discover the range of these DMVs by examining their names, by using the following query:

```sql
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’d typically have to run SQL Trace for a given duration and then spend often a considerable amount of time analyzing and aggregating the results, this made more difficult because the parameters for the same queries would often differ. The corresponding work using DMVs can often be done in seconds.

**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 this range is matched by depth too. DMVs allow you to view a great deal of internal SQL Server information, which 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’ll discuss each of these. Note that DMVs aren’t 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.

**Diagnosing problems**

Diagnosing problems is concerned with identifying the underlying cause of the 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 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 they don’t cause anyone any concern. So while
we could fix them, it would be more appropriate to target our 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, 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 or table scans are being used), and if the statistics are out of date. We can also use the DMVs to identify any missing indexes and 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.

One final point: sometimes if a query is too complicated and contains lots of functionality, then 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 could generate a better plan.

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, and the like. 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 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 two times indicates that a high amount of waiting/blocking may be occurring. Similarly, DMVs also record the amount of I/O (reads/writes at both the physical and logical level), which can be used to measure the effectiveness of a query.

We can examine the cached plan after the improvements have been made to determine whether the bottleneck has been removed. 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, because we should always aim to fix 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 because complexity is often increased. As Knuth pointed out, premature optimization is the root of all software evil.

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
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 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 semipermanent tables).

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.

While you might know which stored procedure is currently running on your server (from your overnight batch scheduler or sp_who2), do you know which 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 that this example uses routines I’ve created and fully documented in the web links given in the code sample in listing 1 (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. This way you’ll be able to adapt this simple monitor pattern and possibly replace the two utilities with your favorite utilities. Later I’ll go through the code that forms the basis of one of stored procedures (dba_WhatSQLIsRunning). Listing 1 shows the code for a simple monitor.

Listing 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_WhatSQLIsRunning
END

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

#A Wait until 7 p.m.
#B Is anything blocked?
#C If blocking occurring...
#D Show SQL running
#E Wait 15 seconds
#F Repeat (500 times)

© Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
This code snippet waits until a specified time (7 p.m. in this example) and then prints the date/time and runs a routine called `dbo.dba_BlockTracer`. If anything is blocked, `dbo.dba_BlockTracer` displays information about both the blockers and the blocked items. In addition, if anything is blocked (and output produced), the variable `@ROWCOUNT` will have a nonzero value; this causes it to output the date and time and list all the SQL that’s running (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) 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).

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, called 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.

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` statement. If `GO` is followed by a number, then it will execute that number of times. This is handy in many...
circumstances; for example, if after an INSERT statement 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 all windows run at the same time, you'll discover the effect of running the SQL at the same time.

It's possible to determine what's running irrespective of any blocking by using an even simpler monitoring query, given in listing 2.

### Listing 2 An even simpler monitor

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

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

SELECT GETDATE() AS TIME
EXEC master.dbo.dba_WhatSQLIsRunning  #B
WAITFOR DELAY '00:00:15'  #C
GO 500  #D
```

#A Wait until 7 p.m.
#B Show running SQL
#C Wait 15 seconds
#D Repeat (500 times)

The query waits for a given time (7 p.m.), displays the date and time together with details of which SQL queries are running, and then waits for a specified period (15 seconds) and repeats (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. We can query the DMVs to determine which queries are waiting (being blocked) the most and aim to improve them.

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

### 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.
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’s often the case that we 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 (many other filters can be applied). Bear in mind that the purpose of these samples is to illustrate quickly how much useful information is freely and easily available within the DMVs.

**Find your slowest queries**

Does anyone ever complain that their 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 listing 3, you’ll identify the 20 slowest queries on your server.

**Listing 3 Find your slowest queries**

```sql
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20
  CAST(total_elapsed_time / 1000000.0 AS DECIMAL(28, 2)) #A
  AS [Total Elapsed Duration (s)],
  execution_count,
  SUBSTRING(qt.text,
    (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 #C
```

#A Get query duration
#B Extract SQL statement
#C Sort by slowest queries

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
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 (`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, 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 query’s cached plan is also output, as XML. The results are sorted by the `total_elapsed_time`. 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 2.

![Figure 2](image-url) I identified the slowest SQL queries on my 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.

The cached plan is probably the primary resource for discovering why the query is running slowly and often will give an insight into how the query can be improved.

The NULL values in the `DatabaseName` column mean the query was run either ad hoc or using prepared SQL (not as a stored procedure). This 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.

**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

©Manning Publications Co. Please post comments or corrections to the Author Online forum: [http://www.manning-sandbox.com/forum.jspa?forumID=XYZ](http://www.manning-sandbox.com/forum.jspa?forumID=XYZ)
have been created. Running the SQL query given in listing 4 will identify the top 20 indexes, ordered by impact (Total Cost), that are missing from your system.

**Listing 4 Finding missing indexes**

```sql
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20
    ROUND(avg_total_user_cost * avg_user_impact *
        (user_seeks + user_scans),0) AS [Total Cost]
    , 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
```

#A Calculate cost

#B 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 (which database/schema/table it applies to, which columns the index would include). These two DMVs (metrics and names) are linked together via another DMV, `sys.dm_db_missing_index_groups`.

I should note how the Total Cost of the missing index is calculated. 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.

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 3.
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 to have included at the leaf level for quicker access.

**Identify which SQL statements are running now**

Often you may know that a particular batch of SQL (or stored procedure) is running, but you don’t 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 5. 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 the 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 we ignore any system processed having a spid of 50 or less, and we ignore this batch too.

**Listing 5 Identifying what SQL is running now**

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum: [http://www.manning-sandbox.com/forum.jspa?forumID=XYZ](http://www.manning-sandbox.com/forum.jspa?forumID=XYZ)
SELECT session_Id AS [Spid],
       ecid,
       DB_NAME(sp.dbid) AS [Database],
       nt_username,
       er.status,
       walt_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 - er.statement_start_offset)/2)
                   + 1) AS [Individual Query],
       qt.text AS [Parent Query],
       program_name,
       Hostname,
       nt_domain,
       start_time
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

#A Extract SQL statement
#B Join request to sysprocesses

The DMV sys.dm_exec_requests contains details of each request, the SQL query, executing on SQL Server. This DMV is joined to the catalog view sys.sysprocesses based on the session id. Catalog views are similar to DMVs but contain static data. The catalog view sys.sysprocesses contains information about the environment from which the request originated, including 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 applying 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 4.

![Figure 4 Output identifying which SQL queries are currently running on the server](http://www.manning-sandbox.com/forum.jspa?forumID=XYZ)
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’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 (SSMS). 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 whether the statistics need to be updated).

But there are problems with this approach. If the query contains temporary tables, getting the estimated plan won’t work; because the query isn’t executed, the temporary table never gets any rows for subsequent processing. In addition, 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 need the relevant SQL to retrieve it using the DMVs. If you run the SQL query given in listing 6, 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, 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. We also ignore this batch of SQL too, by filtering out its spid.

Listing 6 Quickly find a cached plan

```sql
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'
```

(why it’s waiting), the hostname, the domain name, and the start time (useful for determining when the batch was started).

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
AND st.text LIKE 'CREATE PROCEDURE'  

#A Join cached plan and SQL text DMVs  
#B Text to search plan for

Running the Quickly Find a Cached Plan query on my server gives the results shown in figure 5.

![Figure 5 Output showing searched-for cached plans](image)

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 column named query_plan will display the query plan. How it does this differs depending on whether you’re using version 2005 or 2008 of SQL Server. 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, 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 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’re given at the top of each section in green, with text starting “Missing Index” (again see figure 6). If you right-click the diagram, you can select Missing Index Details. Clicking this will open a new window with the definition of the missing index, ready to add; you need to add an appropriate index name. An example of this is shown in listing 7.

**Listing 7 Missing index details**

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

USE [YourDatabaseName]
GO
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=XYZ
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 called SwapsDiaryFile, I can quickly get its cached plan, part of which is shown in figure 7.

Figure 7 Cached plan showing cost by statement and within each statement
Looking at figure 7, you can see that each statement within a batch or stored procedure has a query cost associated with it (here you’ll notice 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 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 identify the section within the batch that should be targeted for improvement.

**Categories of DMVs**

Most sources categorize DMVs in the same manner that Microsoft has adopted, based on their area of functionality. I realize this might not always be the optimal method when researching how to fix problems; bearing this in mind we’ll construct an alternate table of contents, which is structured by problem area.

A brief outline of each of the DMV categories follows in table 1.

Table 1 The major DMV groups

<table>
<thead>
<tr>
<th>DMV Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Data Capture</td>
<td>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.</td>
</tr>
<tr>
<td>Common Language Runtime</td>
<td>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 and smaller than a session), loaded assemblies, properties, and running tasks.</td>
</tr>
<tr>
<td>Database Mirroring</td>
<td>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.</td>
</tr>
<tr>
<td>Database</td>
<td>These DMVs contain information relating to various aspects of databases, including space usage, partition statistics, and session and task space information.</td>
</tr>
<tr>
<td>Execution</td>
<td>These DMVs contain information relating to various aspects of query execution, including cached plans, connections, cursors, plan attributes, stored procedure</td>
</tr>
</tbody>
</table>
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 (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 take 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 the 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 the service broker, including activated tasks, forwarded messages, connections, and queue monitors.

SQL Server Extended  
Extended events allows SQL Server to integrate into Microsoft’s wider event-handling processes, allowing integration of SQL Server events with logging and monitoring tools.

SQL Server Operating  
These DMVs contain information relating to various aspects of the SQL Server
System 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).

Summary
This short introduction to Dynamic Management Views has illustrated the range and depth of information that’s available quickly, easily, and freely, all yours for the asking.

We’ve discussed 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.

I’ve provided various example SQL snippets and discussed them. These should prove immediately useful in determining your slowest SQL queries, identifying your mostly costly missing indexes, identifying which SQL statements are running on your server now, and retrieving the cached plan for an already executed query. In additionally, I’ve provided a useful simple monitor.