WRITTEN BY Michael K. Campbell

ESSENTIAL PERFORMANCE TOOLS FOR SQL SERVER DBAS

Optimizing SQL Server performance can be a daunting task. Especially so for an increasing number of reluctant DBAs faced with the task of managing their organization's SQL Server databases, despite little training or prior experience. The purpose of this white paper is to provide a high-level overview of core SQL Server performance concepts along with an overview of four essential SQL Server performance tools that every DBA should know about.

An Overly Simplified Approach to SQL Server Performance

At a very high and overly simplified level, SQL Server performance can typically be distilled down to a question of how well SQL Server is able to utilize Physical Memory (or RAM). For example, a server with a single, 20GB, OLTP database running on a SQL Server with 32GB of RAM will typically perform well – because it's typically able to keep the entire database in memory. However, if that same database grows to over 40GB in size, then it's possible for serious performance problems to develop as the disk sub-system (which is exponentially slower than RAM) becomes taxed with keeping more data available to end-users than can be comfortably stored in RAM.

With this overly simplified perspective, operations remain performant when the amount of regularly queried data is kept smaller than the amount of available memory. In many cases, this happens naturally. For example with a 120GB OLTP database, end users might only regularly query data from an 'active' portion of the database that is only, say, 20GB in size.

As such, many SQL Server deployments are on powerful enough hardware to mask a host of potentially serious performance problems until databases begin to 'outgrow' underlying hardware. And, obviously, poor coding practices, or the presence of certain kinds of operations, can negatively impact performance with databases of any size – regardless of available RAM or other hardware.

However, trying to keep databases small enough to fit within available memory won't work with many production systems – which are where indexes come into play.

The Power of Indexes

Ignoring, for a moment, the structural characteristics of indexes, a key benefit of indexes is that they're typically much smaller to search than entire tables. For example, a relatively narrow table with 14 million rows that requires a bit more than 8GB of disk of storage will take about 1-2 minutes to execute the query below (assuming 80-120MB/sec of sustained reads from the disk sub-system).

```
SELECT
       CommentId,
       VideoId,
       MemberId,
       Comment,
       Rating,
       CommentTime
FROM
       dbo.Comments
WHERE
       VideoId = 72994
```

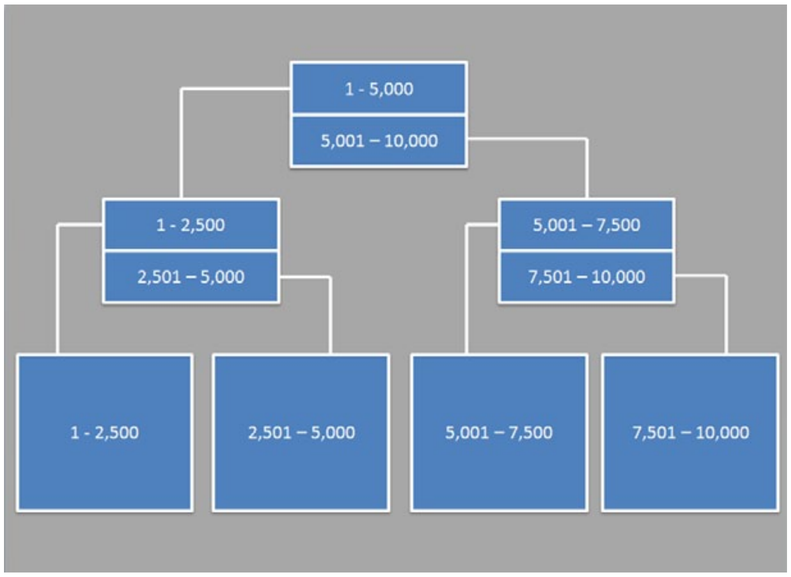


Figure 1. A vastly over-simplified b-tree shows how indexes speed access to data

On the other hand, an index on the VideoId column (since it is the column being queried), would only require around 200MB of storage– a substantial improvement. However, because indexes intelligently store and organize data in a highly optimized fashion using Balanced Trees (or b-trees[1]) which enable SQL Server to quickly 'seek' its way to data using highly optimized algorithms, indexes provide exponentially bigger benefits than a mere reduction in storage.

In fact, to get a sense of just how efficient indexes are, it would be safe to assume that searching a roughly 200MB index should only take 1-2 seconds on systems with 80-120MB/sec of throughput on the disks. Instead, the query above takes less than 10 milliseconds to execute and only reads in 32KB of data from disk or RAM when using an index.

In effect, indexes facilitate quick 'seeks' against data – in much the same way that an index at the back of the book makes it easier to look up specific topics instead of having to 'scan' through an entire book word-by-word. Of course, there is a bit of performance overhead that comes into play as SQL Server does a 'lookup,' known as a key lookup or bookmark lookup in SQL Server parlance, of the data after finding where it should be located from the index, but the net effect of using indexes remains positive.

With SQL Server, it's also possible to create indexes that span multiple columns. For example, in the following query, an index could be created to 'cover' both the VideoId and the MemberId together, which would drastically boost performance.

---

[1] **For more information on Balance-Trees in General, see:**
http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree.
For more information on b-trees and Indexes within SQL Server, see:
http://msdn.microsoft.com/en-us/library/aa964133(SQL.90).aspx
http://www.sqlteam.com/article/sql-server-indexes-the-basics

```
SELECT
       CommentId,
       ... same as before ...
       CommentTime
FROM
       dbo.Comments
WHERE
       VideoId = 72994 AND MemberId = 345
```

Likewise, by 'including' all of the columns needed to satisfy this query, such as the CommentId, Comment, Rating, and CommentTime, SQL Server is able to create what is known as a Covering Index, which puts a copy of all of the data needed to satisfy the query right into the index itself.

For queries that need to perform 'ranged' operations or lookups – such as queries that need to pull back data from a table for a particular date range – SQL Server offers the ability to define a single Clustered Index per table which, defines how data is logically sorted or stored within a table. For example, if a phone book were compared to a table, a clustered index for the phone book would correspond to how entries are logically sorted sometimes by town or city, and then by last name and first name. In this way, a ranged query would be analogous to returning all people with the last-name of 'Smith' in a given town. On the other hand, if SQL Server needed to pull back all users with the first-name of 'John,' the clustered index ends up being useless because SQL Server has to scan each entry in the phone book and evaluate whether or not each person's first name is 'John'.

Another approach to handling queries for 'John' in the phone book would be to create a new index in the back of the phone book that listed everyone by their first name. Only, the problem with this approach is that the lookups needed to go from the index to the back of the phone book for each matching person by the name of 'John' to their existing entry in the 'normal' part of the phone book would create entirely too many 'bookmark lookups.' As such, a better approach would be to create a 'covering index' in the back of the phone book, where each person was listed as first name, last name, and then phone number. In this way, queries by first name would become as easy to process as queries by last name – because a specialized, covering, index would exist specifically for 'first-name' queries.

The only issue with such an approach would be the overhead of keeping the copies of data in the index whenever changes occurred. Typically, this maintenance overhead is negligible, owing largely to the efficient nature of indexes. But, over time, indexes can become fragmented, which can induce performance degradation. It's also possible, yet relatively rare, for too many indexes to be created, or for an index on an extremely large table to cause heavy 're-indexing' when data is repetitively moved or added.

Essential Tools for Finding Performance Problems

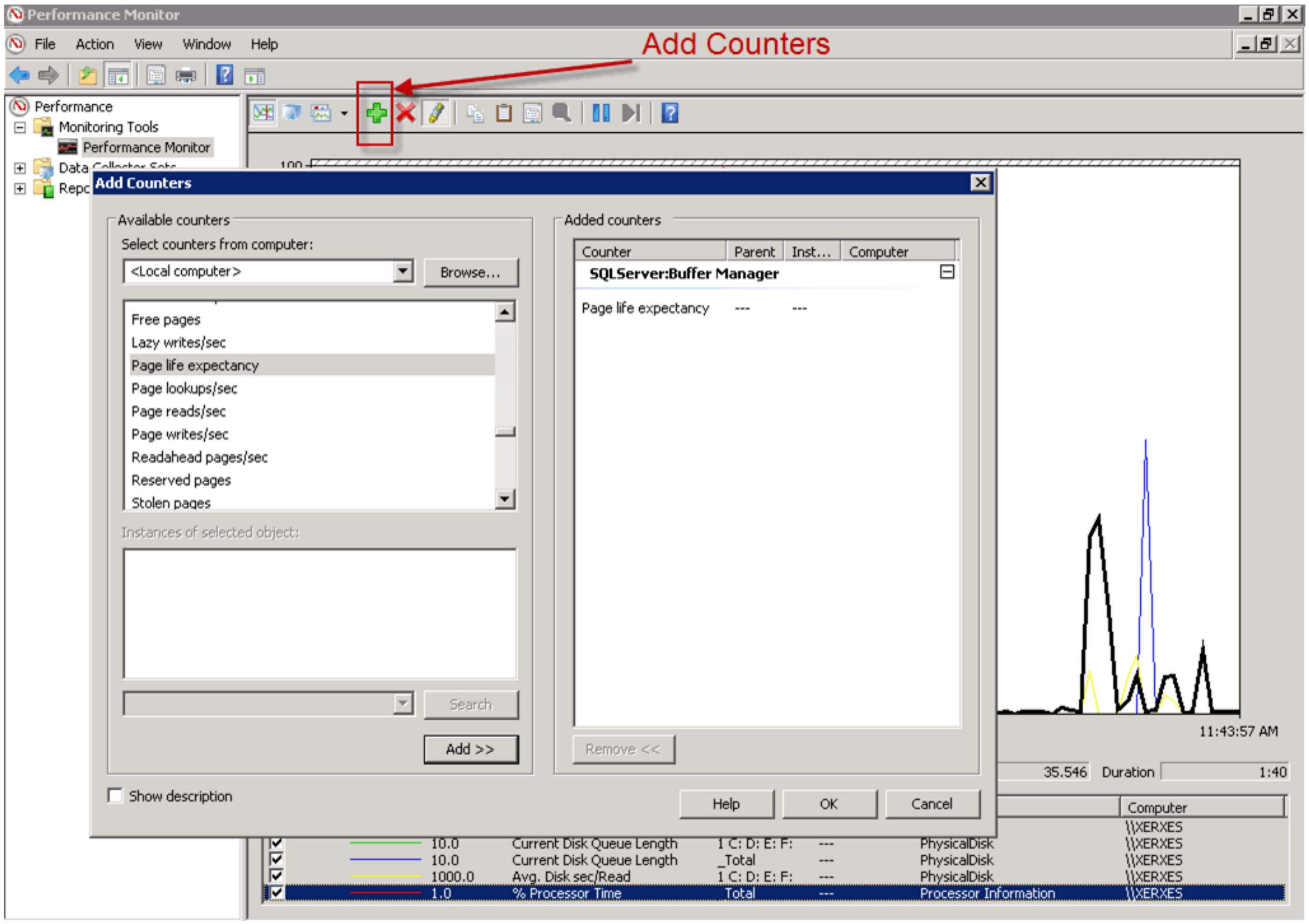
Once the benefits of indexes are understood, performance problems largely become a question of determining whether existing indexes are performing well enough, ensuring that code is able to leverage indexes properly and avoiding worst practices, determining where to add new indexes, or discerning if more hardware is needed to handle, effectively, optimized workloads. Happily, tackling these tasks is made possible with very powerful tools that professional SQL Server DBAs use on a regular basis to diagnose, analyze, and correct SQL Server performance problems.

Performance Monitor (Perfmon)

Performance Monitor, commonly called perfmon, ships with Windows and provides instrumentation for low-level operations at the hardware and operating-system level. In effect, code executed by the Windows operating system has been designed to increment or decrement counters that keep tabs on operations such as CPU utilization, memory usage, or interactions with the disk sub-system. For example, each time the disk sub-system pulls in data from the drives, instrumentation can keep track of how many bytes/second were pulled in, how many operations per second were executed, how many times the IO sub-system queued, how long each physical operation against the drives took, and so on. Even better, since these performance counters are a core part of how Windows works, developers can use them to instrument their own. Happily, SQL Server does a fantastic job of exposing its core metrics via these same counters – which means that perfmon can be used to view, analyze, and log performance counters.

To launch Performance Monitor, just type 'perfmon' into the **Start > Run** dialog. It can also be accessed from the **Administrative Tools** folder. Once launched, you can get a sense for how it works by looking at the **Performance Monitor** node (or **System Monitor** node on Windows Server 2003) and watching the **% Processor Time** counter, which tracks overall processor usage on the system (just as you see when looking at the **Performance** tab in **Task Manager**).

To add additional counters, just click on the plus (+) icon above the graph, and you'll be presented with a dialog that lets you select which computer to add counters from, along with counter families or groups and specific counters (Figure 2).



Select counters from computer: <Local computer>

Free pages
Lazy writes/sec
Page life expectancy
Page lookups/sec
Page reads/sec
Page writes/sec
Readahead pages/sec
Reserved pages
Stolen pages

Counter	Parent	Inst...	Computer
SQLServer:Buffer Manager			
Page life expectancy	---	---	

35.546 Duration 1:40

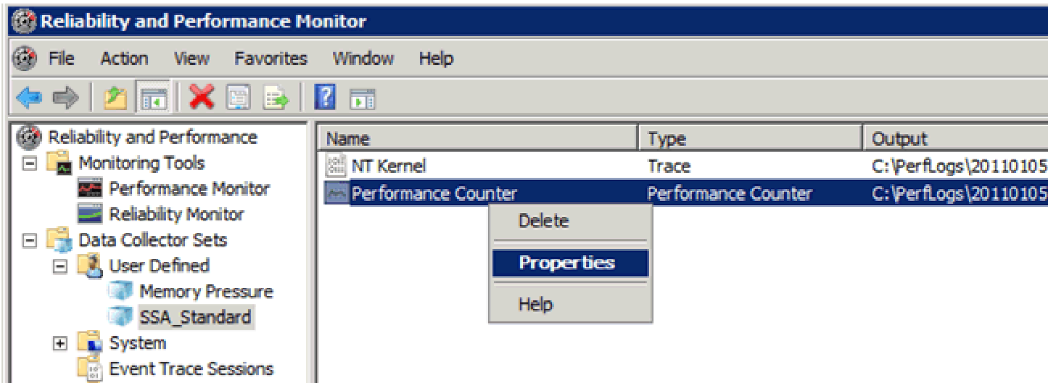
							Computer
✓	10.0	Current Disk Queue Length	1 C: D: E: F:	---	PhysicalDisk		\\XERXES
✓	10.0	Current Disk Queue Length	_Total	---	PhysicalDisk		\\XERXES
✓	1000.0	Avg. Disk sec/Read	1 C: D: E: F:	---	PhysicalDisk		\\XERXES
✓	1.0	% Processor Time	_Total	---	Processor Information		\\XERXES

Figure 2. Performance Monitor – adding new counters

With some counters you will also need to specify which instances of a counter to track (such as with disks, do you want to watch all physical disks, one or two physical disks, or just one for example). Then, in graphing mode, if you click on the **Highlight** icon, whichever counter you have selected in the list of counters below the graph (**% Processor Time** in Figure 2) gets highlighted in the graph, making it more visible.

In addition to monitoring counters in real time, it's also possible to define sets of counters that can be scheduled for collection and logged to disk for later review. Scheduling the collection of performance counter data can be used to gather snapshots of data at peak, non-peak, or other times, making perfmon a great tool for establishing baselines that can be compared against data collected at later dates. Likewise, baselines can also be compared with newly collected data after performance tweaks or development changes have been put into play.

Scheduling the collection of data is fairly easy but you'll want to try it out a few times before you get the hang of it. With Windows 2003 and below, you can create and schedule new Counter Logs from the Performance Logs and Alerts node in the perfmon MMC. Whereas with Windows Server 2008 and above, you have to create new Collector Sets, requiring you create the set, then right-click on the Performance Counter component to specify exactly which counters you wish to track (Figure 3).



Name	Type	Output
NT Kernel	Trace	C:\PerfLogs\20110105
Performance Counter	Performance Counter	C:\PerfLogs\20110105

Figure 3. Specific Counters are chose in the 'Performance Counter' object of Collector Sets

Once you've scheduled the collection of Data Collector Sets or Counter Logs, you can double-click on the generated log file (commonly a .blg file) to open it up in the perfmon UI. If a lot of data has been logged, it can take a while for your system to load everything into the graph for viewing. Then you're free to highly specific counters and zero in on Max, Min, and Average values to look for potential bottlenecks or problems.

The Watcher Effect

When working with perfmon (and other performance tools), it's important to remember that monitoring incurs a small amount of performance overhead, which is commonly referred to as the 'watcher effect.'

Be careful of this negative performance overhead when profiling systems that are under extreme load.

Determining which performance counters to monitor is part science, part skill/experience, and part dark art. This isn't helped by the fact that there is very little in the way of in-depth documentation from Microsoft about what each performance counter means – though some sources to consult are included in the footnotes[2].

Otherwise, one key thing to be aware of when using Performance Monitor is that it's very easy to mistake the symptoms or side effects of a hardware bottleneck as being the actual cause of performance problems. For example, a problem with CPU utilization could be caused by excessive use of **% Privileged Time** in the Processor as the system is spending too much time pulling data from disk, which is a 'privileged' operation for the CPU. However, the bottleneck in this case isn't the CPU, and might not even be the disk – especially if SQL Server is pulling data from disk so frequently because there isn't enough RAM to hold on to it long enough for normal operations.

[2] One online reference that is definitely dated (and not specific to SQL Server) can be found here:
http://msdn.microsoft.com/en-us/library/ff647791.aspx#scalenetchapt15_topic9

Likewise, mention of key counters can be found at either of the following links:
http://sqlcat.com/top10lists/archive/2007/11/21/top-sql-server-2005-performance-issues-for-oltp-applications.aspx
http://sqlcat.com/top10lists/archive/2007/11/21/top-10-sql-server-2005-performance-issues-for-data-warehouse-and-reporting-applications.aspx

Other references can be found online – just be aware that many might contain dated (or in some cases) erroneous information. In general, books tend to be much better references than online documentation. To that end, William R. Stanek's Windows Server 2008 Administrator's Pocket Consultant provides a good overview of core OS and system-level counters, while Wrox's Proffessional SQL Server 2008 Internals and Troubleshooting is a fantastic reference in the sense that it provides great context and background information about core system and SQL Server counters – even if it isn't exhaustive in its coverage of all available counters.

As a result, until you become more familiar with performance counters, the best thing to do with perfmon data is to avoid jumping to any conclusions about what you find. Instead, use data that you've collected as a way to identify potential problems to analyze further or as a way to tell you where and what while troubleshooting performance problems.

SQL Server Profiler

Just as Performance Monitor enables analysis of potential pain points at the system or hardware level, SQL Server also provides its own, powerful, monitoring solution that enables extensive instrumentation of what's going on with a given SQL Server instance via a tool known as SQL Server Profiler. Only, unlike Perfmon, SQL Server Profiler enables much more verbose instrumentation in the sense that it's not limited to monitoring only counters. Instead, Profiler is also able to record or expose the exact syntax of the operations being monitored, making it very helpful for performance tuning and troubleshooting.

So, for example, on a system with a high degree of processor utilization that has been traced back to SQL Server, it's possible to set up a SQL Server Profiler trace that captures not only CPU metrics on all queries being executed, but the text of those queries as well. This, in turn, makes Profiler a powerful tool for finding specific performance culprits once a general area of concern has been identified.

However, while Profiler is commonly used to watch for problematic queries, it can also be configured to watch for blocking, deadlocks, page-splits, query compilation, and a host of other specific tasks. More importantly, Profiler also offers complex filtering capabilities that make it possible to watch for operations by specific users, or applications or against specific databases, tables or indexes.

For SQL Server 2005 and above, Profiler is found in the **Performance Tools** folder for SQL Server (Figure 4). (On SQL Server 2000, it's mixed in with all the other tools).

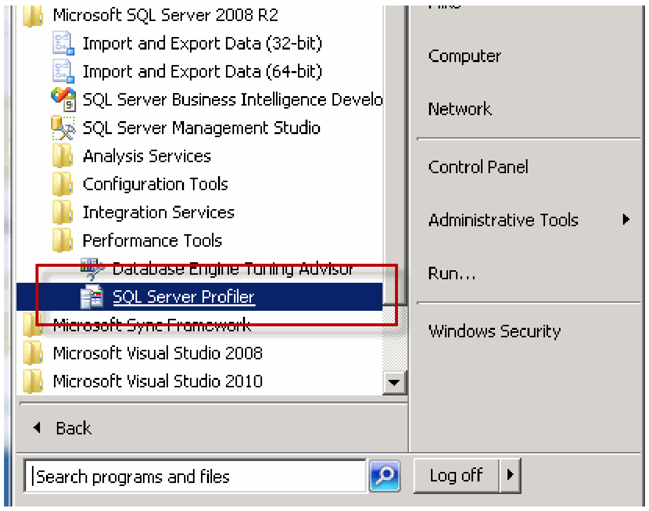

Microsoft SQL Server 2008 R2
Import and Export Data (32-bit)
Import and Export Data (64-bit)
SQL Server Business Intelligence Develo
SQL Server Management Studio
Analysis Services
Configuration Tools
Integration Services
Performance Tools
Database Engine Tuning Advisor
SQL Server Profiler
Microsoft Sync Framework
Microsoft Visual Studio 2008
Microsoft Visual Studio 2010
Back
Search programs and files
Computer
Network
Control Panel
Administrative Tools
Run...
Windows Security
Log off

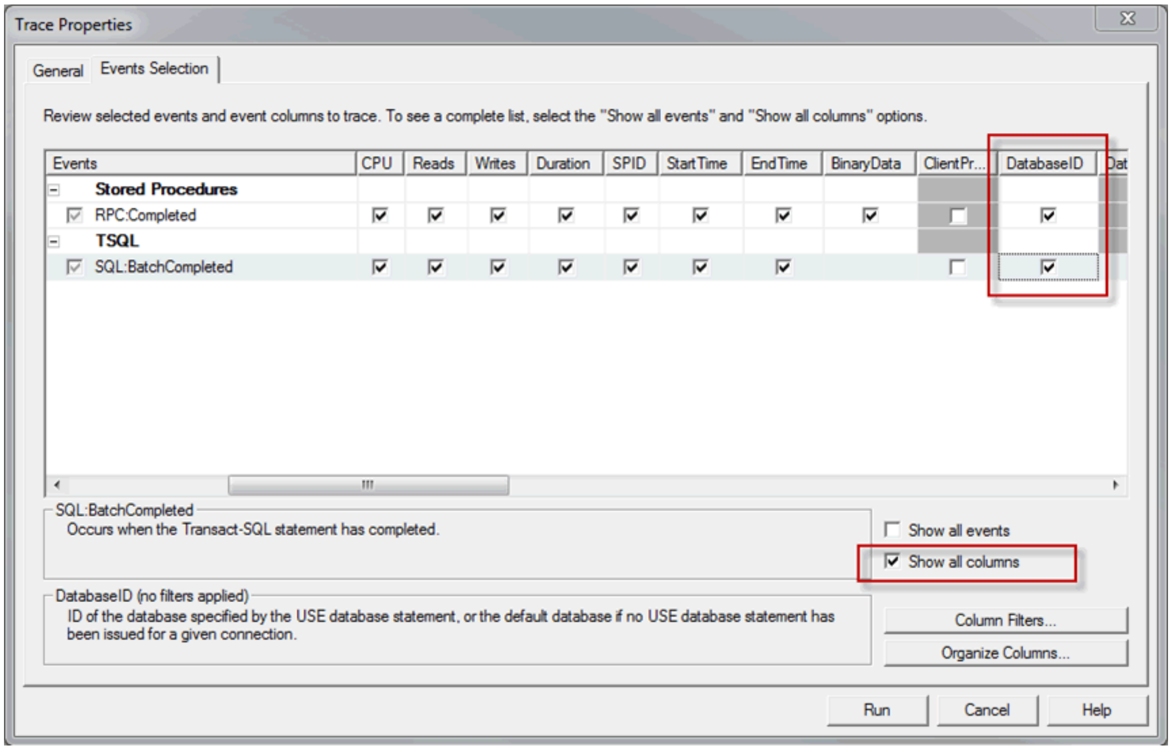
Figure 4. Launching SQL Server Profiler

Once launched, a new trace is created by selecting **File > New Trace** – at which point you'll be prompted to connect to the server you wish to profile. You then need to specify **Trace Properties** – in terms of where to store trace data, and how long to run the trace if you wish it to run unattended. It's also possible to use, or even create, different trace templates, which are a great way to get a quick feel for just how versatile Profiler can be.

However, the key part of setting up any trace is typically focused on which events will be selected or chosen for monitoring along with defining which columns, or data, should be collected with each associated event when it occurs (Figure 5).

As with Perfmon, usage of SQL Server Profiler is susceptible to a 'watcher effect,' especially if you're logging the captured back to a table on the server being profiled. Consequently, logging data to the server being profiled is strongly discouraged as it can create a 'feedback' effect that can negatively impact performance and analysis. Accordingly, I almost always capture trace data to disk as .trc files, which can be double-clicked to open (back in Profiler) once copied to a non-production server or workstation.

Once data from a Profiler trace has been captured, and/or loaded, into SQL Server Profiler, I prefer to save trace data to a table (using the **File > Save As > Trace Table** option) rather than try to slice and dice data within Profiler itself. This way, I can then analyze it with various queries via T-SQL.



Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	CPU	Reads	Writes	Duration	SPID	StartTime	EndTime	BinaryData	ClientPr...	DatabaseID
Stored Procedures										
RPC:Completed	✓	✓	✓	✓	✓	✓	✓	✓	☐	✓
TSQL										
SQL:BatchCompleted	✓	✓	✓	✓	✓	✓	✓		☐	✓

SQL:BatchCompleted
Occurs when the Transact-SQL statement has completed.

☐ Show all events
✓ Show all columns

DatabaseID (no filters applied)
ID of the database specified by the USE database statement, or the default database if no USE database statement has been issued for a given connection.

Figure 5. A simple Profiler Trace to capture all query metrics along with the database they're running in

For example, assuming the standard trace listed in Figure 5, which has been saved to a table called **Trace1**), the following query would provide a quick overview of the most expensive queries in terms of Duration, though it could easily be ordered by Reads, Writes, or CPU instead:

```
DECLARE @commmands TABLE (
        cpu int,
        reads bigint,
        writes bigint,
        duration bigint,
        command varchar(200)
)

INSERT INTO @commmands
SELECT
        CPU,
        Reads,
        Writes,
        duration,
        CASE
                WHEN CHARINDEX('@',CAST(TextData as varchar(200))) = 0 THEN
                        CAST(TextData as varchar(200))
                ELSE
                        SUBSTRING(TextData,0,
                           CHARINDEX('@',CAST(TextData as varchar(200))))
        END [command]
FROM
        Trace1
WHERE
        TextData IS NOT NULL
```

```
DECLARE @totalCount int
DECLARE @totalCPU int
DECLARE @totalReads bigint
DECLARE @totalWrites bigint
DECLARE @totalDuration bigint

SELECT
       @totalCount = COUNT(command),
       @totalCPU = SUM(cpu),
       @totalReads = SUM(reads),
       @totalWrites = SUM(writes),
       @totalDuration = SUM(duration)
FROM
       @commmands

SELECT TOP 20
       LTRIM(command) [Sproc Name],
       COUNT(command) [Number of Calls],
       SUM(CPU) [TotalCPU],
       SUM(Reads) [Total Reads],
       SUM(Writes) [Total Writes],
       SUM(Duration) [Total Duration],
       CASE
              WHEN @totalCount = 0 THEN -1
              ELSE CAST(100 * (CAST(COUNT(command) as decimal)
              / CAST(@totalCount as decimal)) as decimal(8,2))
       END [% of Calls],
       CASE
              WHEN @totalCPU = 0 THEN -1
              ELSE CAST(100 * (CAST(SUM(cpu) as decimal)
              / CAST(@totalCPU as decimal)) as decimal(8,2))
       END [% of CPU],
       CASE
              WHEN @totalReads = 0 THEN -1
              ELSE CAST(100 * (CAST(SUM(reads) as decimal)
              / CAST(@totalReads as decimal)) as decimal(8,2))
       END [% of Reads],
       CASE
              WHEN @totalWrites = 0 THEN -1
              ELSE CAST(100 * (CAST(SUM(writes) as decimal)
              / CAST(@totalWrites as decimal)) as decimal(8,2))
       END [% of Writes],
       CASE
              WHEN @totalDuration = 0 THEN -1
              ELSE CAST(100 * (CAST(SUM(Duration) as decimal)
              / CAST(@totalDuration as decimal)) as decimal(8,2))
       END [% of Duration]
FROM
       @commmands
GROUP BY
       command
ORDER BY
       SUM(duration) DESC
```

This query is very rudimentary and ideally suited towards environments where stored procedures are favored over parameterized, prepared, or ad-hoc queries. Consequently, the first part of this query involves simplistic 'normalization' of stored procedures, making it possible to aggregate calls to stored procedures while ignoring their parameters.

For more comprehensive analysis of performance data, a fantastic free tool that many DBAs use is ClearTrace offered by ScaleSQL Consulting[3]. This free tool does an extensive job of normalizing queries and operations, then makes it very easy to 'slice and dice' results in a simple-to-use UI. This is why it's so popular with DBAs as a tool to help them quickly analyze performance traces.

However, the key point of using SQL Server Profiler in the manner outlined (i.e., to review aggregate metrics from executing queries) is to help spot which queries are taking the longest to execute, burning up the most CPU, or causing the most reads/writes. Armed with this information, it's then possible to begin looking at the execution plans for the queries in question to see if they can be optimized.

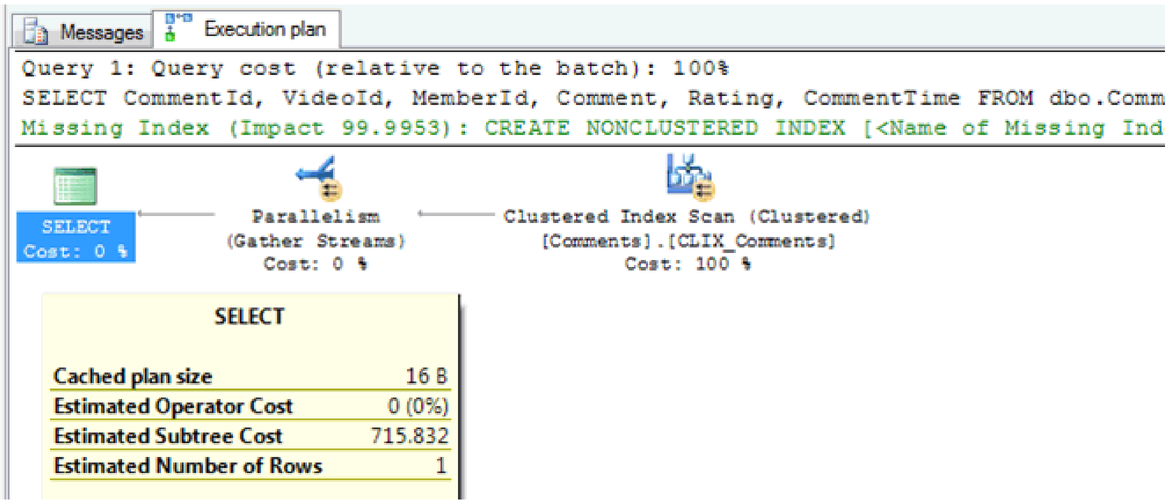
SQL Server Execution Plans

When a query is processed by SQL Server, the Command Parser validates syntax and then passes execution off to the Optimizer, which is tasked with determining an efficient method for retrieving or modifying the data specified in a query. The optimizer, in turn, checks parameterization of the query in question, looks for an existing execution plan that matches the syntax of the query or operation in question, and generates a new execution if needed. Execution plans, in turn, roughly equate to the set of instructions that the Query Executor will use to work with underlying data in order to define the specific, physical, operations that will be executed against tables, and other data structures to perform the operation in question.

The specifics of how all of this transpires within mere milliseconds is outside the scope of this paper. Needless to say, it does occur with complex cost-based algorithms that manage to do a fantastic job of executing queries. The good news is that SQL Server makes the actual execution plans it uses available for DBAs and developers to view and analyze. This, in turn, means that query plans can be easily analyzed and evaluated to locate expensive operations, verify the optimal use of indexes, or tuned for better performance.

While there are many ways to access execution plans and their related details, two basic options are typically used. First, is the estimated execution plan – which is the execution plan that SQL Server will use to execute the query – but using statistical, or sampled, data instead of actual metrics from or about the underlying data to be operated on. Using estimated execution plans can be beneficial in cases where the query might take a long time to run, or where it might adversely impact the system. For example, if you've identified a query that logs orders into the system as a potential performance culprit, you typically don't want to just re-run that query for fear of duplicating orders etc.).

To view a query's estimated execution plan, just press **CTRL+L** (or **CTRL** and **L** at the same time) after selecting the text of the query you want to run. (Or if the query you wish to profile is the only query in the editor, you don't need to select anything. Likewise, instead of using shortcuts for execution plan options, you can use options from the Query menu in SQL Server Management Studio.)



```
Query 1: Query cost (relative to the batch): 100%
SELECT CommentId, VideoId, MemberId, Comment, Rating, CommentTime FROM dbo.Comm
Missing Index (Impact 99.9953): CREATE NONCLUSTERED INDEX [<Name of Missing Ind
```

SELECT

Cached plan size	16 B
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	715.832
Estimated Number of Rows	1

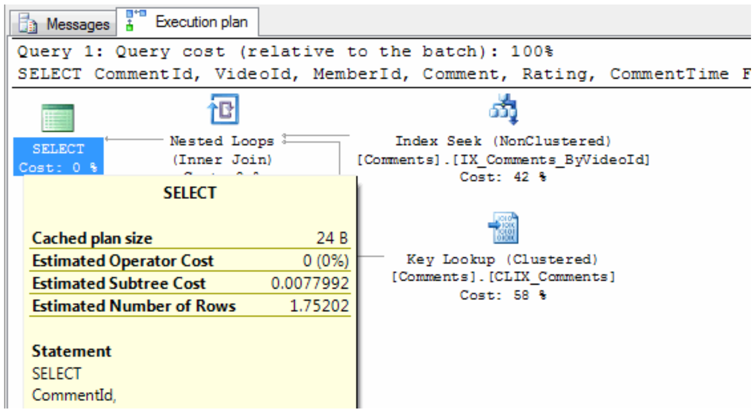
Figure 6. A sample execution plan against a table with no indexes defined

[3] ClearTrace by ScaleSQL Consulting: http://www.scalesql.com/cleartrace/default.aspx

As an example, the query mentioned in the beginning of this white paper is profiled below in Figure 6 – without any indexes defined on the Comments table.

As you can see in Figure 6, not only does the execution plan for this query show that a Table Scan, meaning that SQL Server is reduced to pulling the entire table into memory, and then iterating over every row to evaluate the where clause, was used, but it also shows that SQL Server recognizes that the existence of an index would provide a dramatic boost to performance. Figure 6 also highlights how moving the mouse over individual steps in the plan causes a details popup to be displayed. In this case, the mouse is directly over the final select operation, displaying estimated cost metrics.

As covered in the section on indexing, the addition of a simple index on the VideoId column of the Comments table will have a dramatic impact on the query in question, which SQL Server has recognized as well. Consequently, Figure 7 shows the estimated execution plan of the same query, which has been executed after a simple index was added to the Comments table.



```
Query 1: Query cost (relative to the batch): 100%
SELECT CommentId, VideoId, MemberId, Comment, Rating, CommentTime F
```

SELECT
Cost: 0 %

Nested Loops
(Inner Join)

Index Seek (NonClustered)
[Comments].[IX_Comments_ByVideoId]
Cost: 42 %

Key Lookup (Clustered)
[Comments].[CLIX_Comments]
Cost: 58 %

SELECT

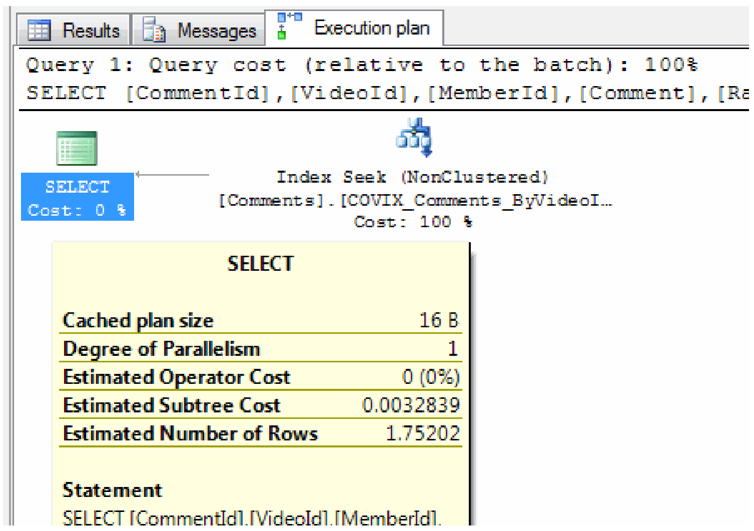
Cached plan size	24 B
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0077992
Estimated Number of Rows	1.75202

Statement
SELECT
CommentId,

Figure 7. The same query but with an index involved

However in this case, a key lookup, also known as a bookmark lookup, is in effect, because the index on VideoId is being used to find which rows satisfy the query, and then those rows are, in turn, being looked up by the Clustered Index on the table itself, meaning that multiple operations are required to pull back the data in question. Regardless, the use of a simple index has resulted in an exponential improvement in performance, moving the cost of this query from 715+ to .007.

On the other hand, Figure 8 provides a final version of the same query. This time, however, a covering index has been added to the Comments table, meaning that SQL Server can perform a single index seek and obtain all of the data needed. This results in a much cleaner operation that ends up being even more performant in that it costs roughly half of what its predecessor did.



```
Query 1: Query cost (relative to the batch): 100%
SELECT [CommentId],[VideoId],[MemberId],[Comment],[Ra
```

SELECT
Cost: 0 %

Index Seek (NonClustered)
[Comments].[COVIX_Comments_ByVideoI...
Cost: 100 %

SELECT

Cached plan size	16 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0032839
Estimated Number of Rows	1.75202

Statement
SELECT [CommentId],[VideoId],[MemberId],

Figure 8. An Actual Execution Plan of the same query but with a covering index involved

Importantly, Figure 8 was obtained by pressing **CTRL+M** and then executing the query rather than just getting the estimated execution plan. Consequently, Figure 8 displays details from an Actual Execution Plan instead of an Estimated Execution Plan, even though the details displayed still reference 'estimated' details. This, in turn, is because SQL Server uses the same execution plan for both actual and estimated plans – it's just the details that end up getting changed.

It's also important to point out that these execution statistics are not always wholly accurate. Consequently, execution plans should really be used to determine the operations that SQL Server is using to physically execute queries. However, since these metrics can and will be inaccurate on occasion, the best thing to do is to open up SQL Server Profiler and trace the query before changes and then after changes and then compare CPU, Duration, Reads, and Writes to ensure that a net benefit has been introduced as desired. This way, you can ensure that some of the changes you have made to a query, such as adding indexes or changing operations within the query, are creating a positive effect.

SQL Server DMVs

Given that SQL Server stores execution plans for internal use, and given that SQL Server can expose all sorts of metrics and instrumentation for use in troubleshooting and tuning, it shouldn't come as much of a surprise that much of this data can actually be queried from within SQL Server itself. For example, it's possible to query performance counters directly from within SQL Server – without using Performance Monitor. Just as it's also possible to set up Profiler traces from directly within SQL Server without the need for SQL Server Profiler.

As such, one of the big changes with SQL Server 2005 was the comprehensive consolidation of these kinds of internal details into entire 'families' of pseudo-tables and other query-able objects, collectively known as DMVs – or Dynamic Management Views, which is a bit of a misnomer as some of these 'views' are functions or other objects.

As outlined in Books Online[4], some DMVs focus on security or storage, while others focus on change data capture or database mirroring. Other DMVs focus on index statistics and management (and can be used to find non-used[5] or missing indexes), while others provide valuable information about execution plans, currently executing operations, or resources that SQL Server is regularly waiting on. Collectively, many of these 'operational' DMVs serve as a sort of 'window' into the workings of the SQL Server engine itself, known as the SQLOS (or SQL Operating System) as of SQL Server 2005 and beyond.

Consequently, many DBAs spend considerable time exploring and fine-tuning scripts that they can use to query DMVs for various insights into SQL Server performance. Some of these are outlined below.

DMVs for Most Expensive Queries

An easy way to see an example of just how beneficial DMVs can be is to see how SQL Server Management Studio (SSMS) takes advantage of them right out of the box, in terms of displaying information about recent, expensive, queries.

To see this functionality in action, you will need to log into a Server from SQL Server Management Studio, right click on the server itself, and launch the **Activity Monitor**. Remember, the watcher effect will impose some performance overhead so be careful on heavily encumbered systems.

Once Activity Monitor launches, take note of how SSMS is displaying performance information pulled from counters that profile the SQLOS itself – as in Figure 9. Then, expand the **Recent Expensive Queries** section, to see how SSMS is querying DMVs to obtain information about expensive queries.

Better yet, right-click on a specific query and you'll then be able to view its actual execution plan if desired which SQL Server pulls from the sys.
dm_exec_query_plan DMV.

Then, for a real test of your blossoming skills, fire up SQL Server Profiler, and capture the actual queries being executed when you open up a new instance of SQL Server Management Studio and launch the Activity Monitor. By doing so, you'll be able to see exactly which DMVs SQL Server is querying to provide you with the data that you're using to monitor performance.

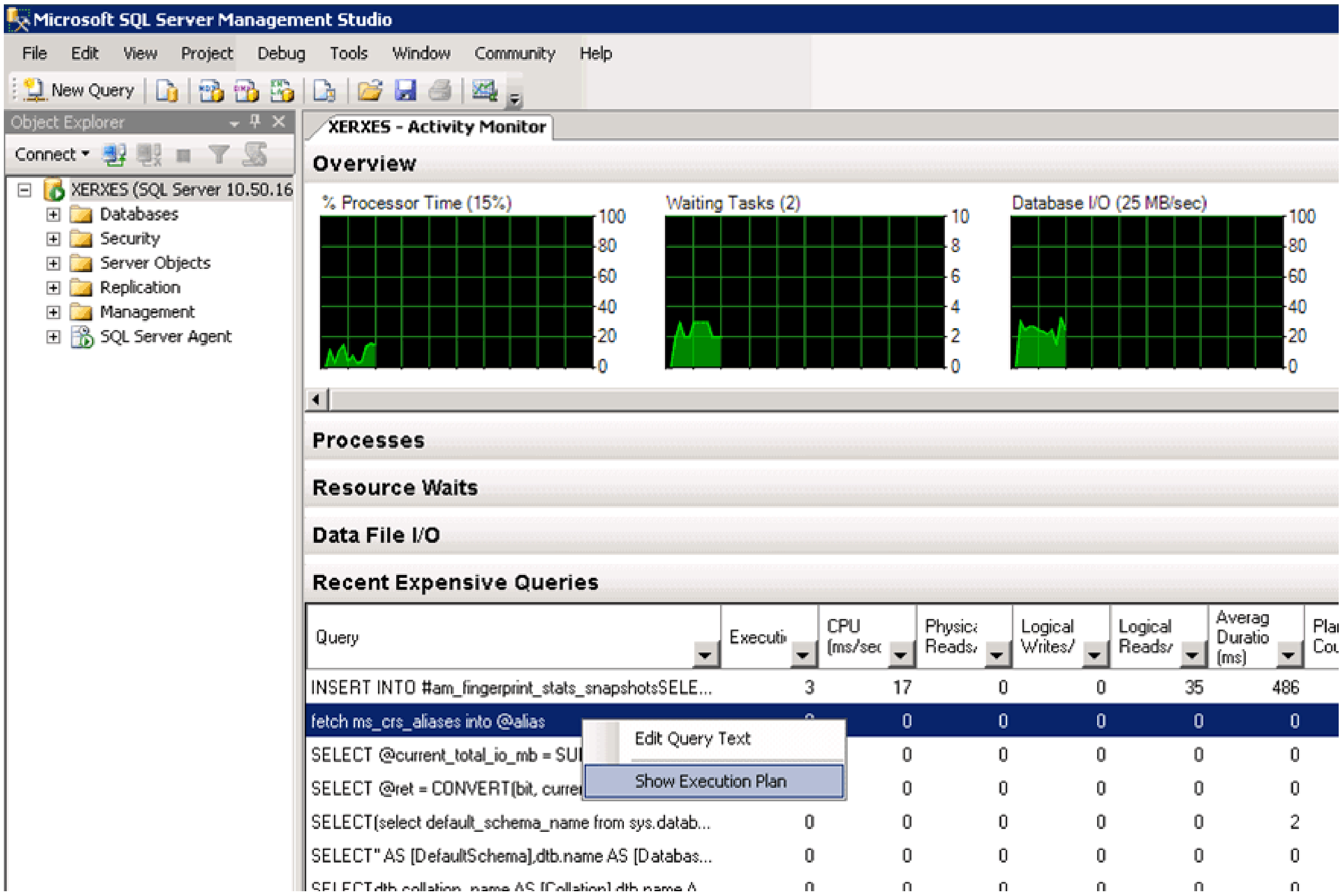
DMVs for WAITS and WAITSTATS

Another powerful use of DMVs comes in the form of allowing DBAs to evaluate SQL Server Wait Statistics – or internal statistics about which resources and operations the SQL Server Engine itself is waiting on. In turn, this data is managed by the SQLOS, which is responsible for dividing up the system resources allocated to SQL Server among the various workers, or sessions, that it uses internally to manage user connections, validate syntax, aggregate and update data, or fetch data from disk. Then, as each worker encounters operations that they must wait on, they return back to the SQLOS's scheduling mechanism, report what they're waiting on, and wait to be scheduled or handed another task. In this way, SQL Server is able to

---

[4] **See http://msdn.microsoft.com/en-us/library/ms188754.aspx for an overview of available DMVs.**

[5] **For an example of how DMVs can be used to find non-used indexes, see the following blog post: http://sqlserverperformance.idera.com/indexing/removing-unused-indexes/**



Query	Executi	CPU (ms/sec	Physica Reads/	Logical Writes/	Logical Reads/	Averag Duratio (ms)	Pla Cou
INSERT INTO #am_fingerprint_stats_snapshotsSELE...	3	17	0	0	35	486	
fetch ms_crs_aliases into @alias	0	0	0	0	0	0	
SELECT @current_total_io_mb = SUI		0	0	0	0	0	
SELECT @ret = CONVERT(bit, curre		0	0	0	0	0	
SELECT(select default_schema_name from sys.datab...	0	0	0	0	0	2	
SELECT'' AS [DefaultSchema],dtb.name AS [Databas...	0	0	0	0	0	0	
SELECTdtb.collation_name AS [Collation],dtb.name A...	0	0	0	0	0	0	

Figure 9. SQL Server Management Studio takes advantage of DMVs to provide performance data

keep very good tabs on the exact kinds of operations and resources that it is waiting in. In fact, by default, SQL Server accumulates these waits from startup.

As such, DBAs can query these wait statistics to gain significant insights into what SQL Server is regularly waiting on. It's also possible to reset wait statistics as well (though this requires a call to a more dated interface in the form of executing DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR) to force a reset of internal wait stats to zero). Consequently, DBAs can gain a sense of what SQL Server might be waiting on over a period of a few minutes or seconds when performance is at its worst or when trying to diagnose a particular problem.

Sadly, a more detailed explanation of how DMVs can be used to query wait statistics is outside the scope of this white paper. However, you can find a footnote reference to an excellent Microsoft white paper by Tom Davidson and Danny Tambs[6], showcasing how wait stats and DMVs can be a powerful tool for discovering insights into SQL Server performance problems.

[6] SQL Server 2005 Waits and Queues by Tom Davidson and Danny Tambs: http://msdn.microsoft.com/en-us/library/cc966413.aspx.

Conclusion / Where to Go From Here?

The purpose of this white paper was not to provide exhaustive coverage of any of the tools or techniques outlined. Instead, this paper provided a high level overview, or survey, of core tools used by SQL Server DBAs to detect, analyze, and correct performance problems. As such, the best way to benefit from the information provided in this white paper is to begin testing and experimenting with the tools outlined. As with any other tool, mastery and expertise of the tools outlined in this paper can only be achieved through hands-on practice and regular use in solving real problems.

However, another alternative for learning about SQL Server performance problems and tuning options is to use third-party solutions which 'wrap' much of the instrumentation and diagnostics r to create comprehensive performance 'dashboard' solutions. For example, Idera, the sponsors of this white paper, offer SQL diagnostic manager, an award-winning tool that allows for easy performance management, monitoring, and troubleshooting. As such, using tools like SQL diagnostic manager represent a great way for new and intermediate DBAs to learn about SQL Server workload characteristics, performance monitoring, and troubleshooting techniques with less initial investment of time and energy by allowing them to experience performance tuning and monitoring from a single, consolidated, interface. Consequently, the use of third-party performance monitoring tools can be a great way to help beginning or reluctant DBAs better come up to speed with performance monitoring techniques and approaches. This means that these solutions typically pay for themselves in multiple ways in short order.

Either way, the best way to learn about performance monitoring, analysis, and tuning is to gain hands-on experience with real production workloads.

ABOUT THE AUTHOR Michael K Campbell (mike@overachiever.net) is a contributing editor for SQL Server Magazine and a consultant with years of SQL Server DBA and developer experience. He enjoys consulting, development, and creating free videos for www.sqlservervideos.com.