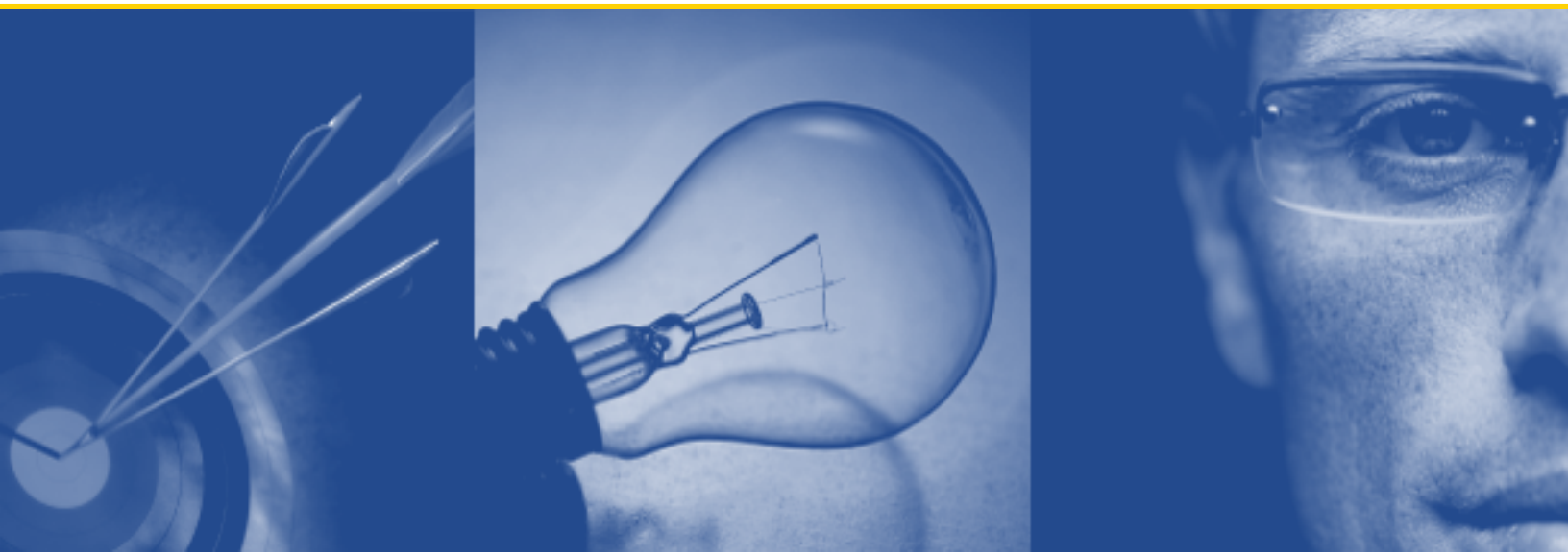
QUEST SOFTWARE®

Top 10 Things You Should Know About Optimizing SQL Server Performance

written by
Patrick O'Keeffe,
Senior Software Architect,
Quest Software, Inc.



World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
e-mail: info@quest.com

Please refer to our Web site for regional and international office information.

Updated—September 17, 2007

CONTENTS

INTRODUCTION

Performance optimization on SQL Server is difficult. A vast array of information exists on how to address performance problems in general. However, there is not much information on the specifics and even less information on how to apply that specific knowledge to your own environment.

In this whitepaper, I discuss the 10 things that I think you should know about SQL Server performance. Each item is a nugget of practical knowledge that can be immediately applied to your environment.

WHAT PROBLEM ARE WE TRYING TO SOLVE?

The problem is a simple one: How do you get the most value from your SQL Server deployments? Faced with this problem, many of us ask: Am I getting the best efficiency? Will my application scale?

A scalable system is one in which the demands on the database server increase in a predictable and reasonable manner. For instance, doubling the transaction rate might cause a doubling in demand on the database server, but a quadrupling of demand could well result in the system failing to cope.

Increasing the efficiency of database servers frees up system resources for other tasks, such as business reports or ad-hoc queries.

To get the most out of an organization's hardware investment, you need to ensure that the SQL or application workload running on the database servers is executing as fast and as efficiently as possible.

There are a number of drivers for performance optimization, including:

- Tuning to meet service level agreement (SLA) targets
- Tuning to improve efficiency, thereby freeing up resources for other purposes
- Tuning to ensure scalability, thereby helping to maintain SLAs into the future

Performance optimization is an ongoing process. For instance, when you tune for SLA targets, you can be 'finished'; however, if you are tuning to improve efficiency or to ensure scalability, your work is never really finished. This tuning should be continued until the performance is 'good enough'. In the future, when the performance of the application is no longer 'good enough', this tuning should be performed again.

'Good enough' is usually defined by business imperatives, such as SLAs or system throughput requirements. Beyond these requirements, you should be motivated to maximize the scalability and efficiency of all database servers—even if business requirements are currently being met.

As stated in the introduction, performance optimization on SQL Server is challenging. There is a wealth of generalized information on various data points (counters, DMVs) available, but there is very little information on what to do with this data and how to interpret it. This paper describes 10 things that will be useful in the trenches, allowing you to turn some of the data into information.

10. BASELINING AND BENCHMARKING ARE A MEANS TO AN END

What Are Baselining and Benchmarking?

Baselining and benchmarking give you a picture of resource consumption over time. If your application has not yet been deployed into production, you need to run a simulation. This can be achieved by:

- Observing the application in real time in a test environment
- Playing back a recording of the application executing in real time

The best outcome is achieved by observing actual workload in real time or playing back a recording of a real time simulation.

Ideally, you would also want to run the workload on hardware comparable to what the application will be deployed on and with “realistic” data volumes. SQL statements that deliver good performance on small tables often degrade dramatically as data volumes increase. The resulting data can then be plotted to easily identify trends.

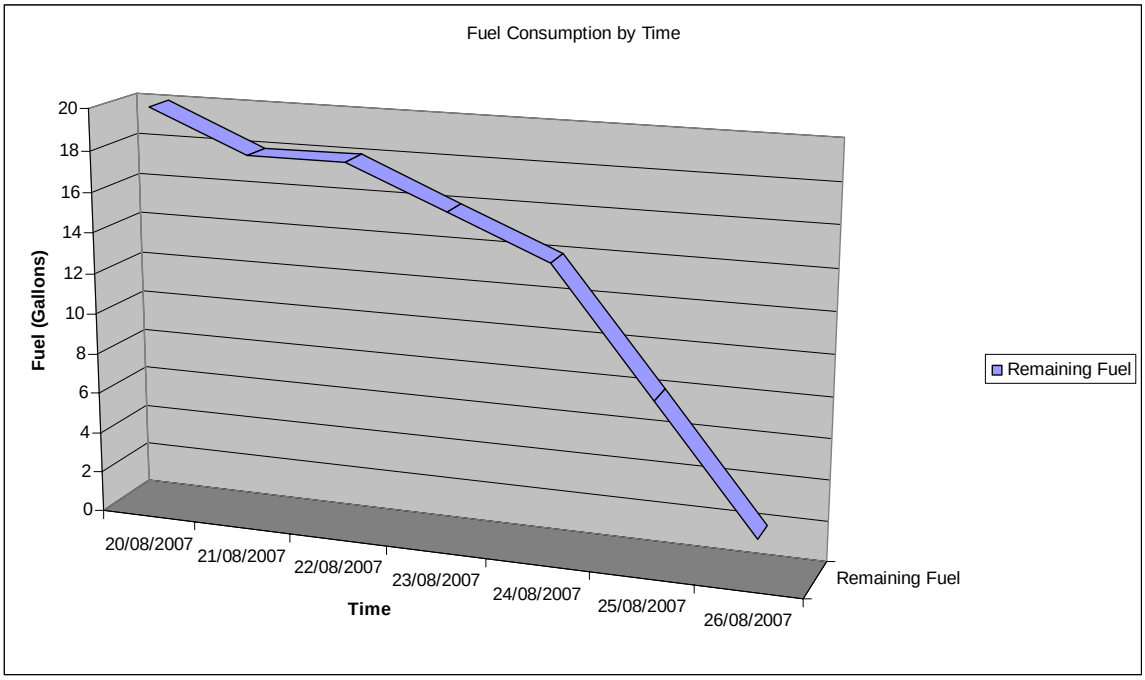
The practical upshot is that you can evaluate future behavior against a baseline, to determine whether resource consumption has improved or worsened over time.

What Baselining Can’t Do

Baselining is not the only tool in your performance optimization toolbox. To explain what baselining and benchmarking can’t do, let’s use the ubiquitous car analogy and talk about fuel consumption.

The performance counter we are going to sample is obviously a fuel gauge.

For the period of a few days, we will sample the level of the fuel in the fuel tank and plot it in a graph shown below.



The plot displays the fuel remaining in the fuel tank over time. We can see that the baseline behavior represents the level of fuel in the tank that decreases slowly at first and then starts to accelerate more quickly towards the end of the measured time period. In general, this is the normal behavior for fuel in a fuel tank over time.

Assuming this graph represents normal behavior, we can measure and plot a different behavior and compare the two graphs. We would easily see the change in behavior. Emerging trends may also be easily identified since we can plot against time.

A baseline cannot, however, provide any qualitative measure of efficiency. From the chart above, you cannot draw any conclusions about how efficient the car is—you must investigate elsewhere for this information. The baseline can tell you only whether you used more (or less) fuel between two days.

Similarly for SQL Server, a baseline can tell you only that something is outside that range of normally observed behavior. It cannot tell you whether the server is running as efficiently as possible.

The point is that you should not start with baselining. You need to make sure that your application workload is running as efficiently as possible. Once performance has been optimized, you can then take a baseline. Also, you cannot simply stop with baselining. You should keep our application running as efficiently as possible and use your baseline as an early warning system that can alert you when performance starts to degrade.

9. PERFORMANCE COUNTERS – HOW TO CUT TO THE CHASE

A very common question related to SQL Server performance optimization is: What counters should I monitor?

In terms of managing SQL Server, there are two broad reasons for monitoring performance counters:

1. Operational
2. Bottlenecks

Although they have some overlap, these two reasons allow you to easily choose a number of data points to monitor.

Operational Monitoring

Operational monitoring checks for general resource usage. It helps answer questions like:

- Is the server about to run out of resources like CPU, disk space or memory?
- Are the data files able to grow?
- Do fixed size data files have enough free space for data?

You also could collect data for trending purposes. A good example would be collecting the sizes of all the data files. From this information, you could trend the data file growth rates. This would allow you to more easily forecast what resource requirements you might have in the future.

To answer the three questions posed above, you should look at the following counters:

COUNTER	REASON
Processor\% Processor Time	Monitor the CPU consumption on the server
LogicalDisk\Free Megabytes	Monitor the free space on the disk(s)
MSSQL$Instance:Databases\Data File(s) Size (KB)	Trend growth over time
Memory\Pages/sec	Check for paging, a good indication that memory resources might be short

Bottleneck Monitoring

Bottleneck monitoring focuses more on performance-related matters. The data you collect helps answer questions such as:

- Is there a CPU bottleneck?
- Is there an I/O bottleneck?
- Are the major SQL Server subsystems, such as the Buffer Cache and Procedure Cache, healthy?
- Do we have contention in the database?

To answer these questions, we would look at the following counters:

COUNTER	REASON
Processor\% Processor Time	Monitor CPU consumption allows us to check for a bottleneck on the server (indicated by high sustained usage).
High percentage of Signal Wait	Signal wait is the time a worker spends waiting for CPU time after it has finished waiting on something else (e.g., a lock, a latch or some other wait). Time spent waiting on the CPU is indicative of a CPU bottleneck. Signal wait data can be found by executing `DBCC SQLPERF(waitstats)` on SQL Server 2000 or by querying `sys.dm_os_wait_stats` on SQL Server 2005.
PhysicalDisk\Avg. Disk Queue Length	Check for disk bottlenecks – if the value exceeds 2^1 then it is likely that a disk bottleneck exists.
MSSQL$Instance:Buffer Manager\Page Life Expectancy	Page Life Expectancy is the number of seconds a page stays in the buffer cache. A low number indicates that pages are being evicted without spending much time in the cache, thereby reducing the effectiveness of the cache.
MSSQL$Instance:Plan Cache\Cache Hit Ratio	A low Plan Cache hit ratio means that plans are not being reused.
MSSQL$Instance:General Statistics\Processes Blocked	Long blocks indicate a contention for resources.

8. WHY CHANGING SP_CONFIGURE SETTINGS PROBABLY WON'T HELP

SQL Server is not like other databases. Very few switches and knobs are available to tweak performance. There are certainly no magic silver bullets to solve performance problems simply by changing an sp_configure setting.

It is generally best to leave the sp_configure settings at their defaults, thereby letting SQL Server manage things. Your time is best spent looking at performance from a workload perspective, such as database design, application interaction and indexing issues.

Let's look at a workload example at a setting and see why it is generally best to leave things alone.

The "max worker threads" setting is used to govern how many threads SQL Server will use. The default value (in SQL Server 2005 on commodity hardware) is 256 worker threads.

This does not mean that SQL Server can have only 256 connections. On the contrary, SQL Server can service thousands of connections using up to the maximum number of worker threads.

If you were responsible for a SQL Server that regularly had 300 users connected, you might be tempted to raise the maximum number of worker threads to 300. You might think that having one thread per user would result in better performance. This is incorrect. Raising this number to 300 does two things:

1. Increases the amount of memory that SQL Server uses. Even worse, it decreases the amount of memory that SQL Server can use for buffer cache, because each thread needs a stack
2. Increases the context switching overhead that exists in all multithreaded software

In all likelihood, raising the maximum number of worker threads to 300 made things worse. It also pays to remember that even in a four-processor box, there can only be four threads running at any given time. Unless you are directed to do so by Microsoft support, it is best to focus your efforts on index tuning and resolving application contention issues.

7. I HAVE A BOTTLENECK – WHAT DO I DO NOW?

Once you have identified a bottleneck and worked out that it is best to leave the sp_configure settings alone, you need to find the workload that is causing the bottleneck.

This is a lot easier to do in SQL Server 2005. Users of SQL Server 2000 will have to be content with using Profiler or Trace (more on that in #6).

In SQL Server 2005, if you identified a CPU bottleneck, the first thing that you would want to do is to get the top CPU consumers on the server. This is a very simple query on `sys.dm_exec_query_stats`:

```
select top 50
       qs.total_worker_time / execution_count as avg_worker_time,
       substring(st.text, (qs.statement_start_offset/2)+1,
   ((case qs.statement_end_offset
       when -1 then datalength(st.text)
      else qs.statement_end_offset
    end - qs.statement_start_offset)/2) + 1) as statement_text,
       *
from
       sys.dm_exec_query_stats as qs
       cross apply sys.dm_exec_sql_text(qs.sql_handle) as st
order by
       avg_worker_time desc
```

The really useful part of this query is your ability to use `cross apply` and `sys.dm_exec_sql_text` to get the SQL statement so you can analyze it.

It is a similar story for an I/O bottleneck:

```
select top 50
       (total_logical_reads + total_logical_writes) as total_logical_io,
  (total_logical_reads/execution_count) as avg_logical_reads,
  (total_logical_writes/execution_count) as avg_logical_writes,
  (total_physical_reads/execution_count) as avg_phys_reads,
  substring(st.text, (qs.statement_start_offset/2)+1,
  ((case qs.statement_end_offset
             when -1 then datalength(st.text)
      else qs.statement_end_offset
   end - qs.statement_start_offset)/2) + 1) as statement_text,
*
from
       sys.dm_exec_query_stats as qs
       cross apply sys.dm_exec_sql_text(qs.sql_handle) as st
order by
       total_logical_io desc
```

6. SQL PROFILER IS YOUR FRIEND

If we have SQL Server 2000 (which means no DMVs to make your life easier), you can still obtain (with a little more work) similar information to identify and classify workload.

In the following step, I use Profiler Traces from a remote machine. There is nothing to stop you from using server side traces instead. The important part is what you do with the raw data once it gets into a database table.

Start a Trace

The goal is to classify workload so I have chosen these four SQL-related events:

- RPC:Completed
- SP:Completed
- SQL:BatchCompleted
- SQL:StmtCompleted

Figure 1 shows the Trace Properties Dialog. I have also chosen all possible columns for each of these event types.

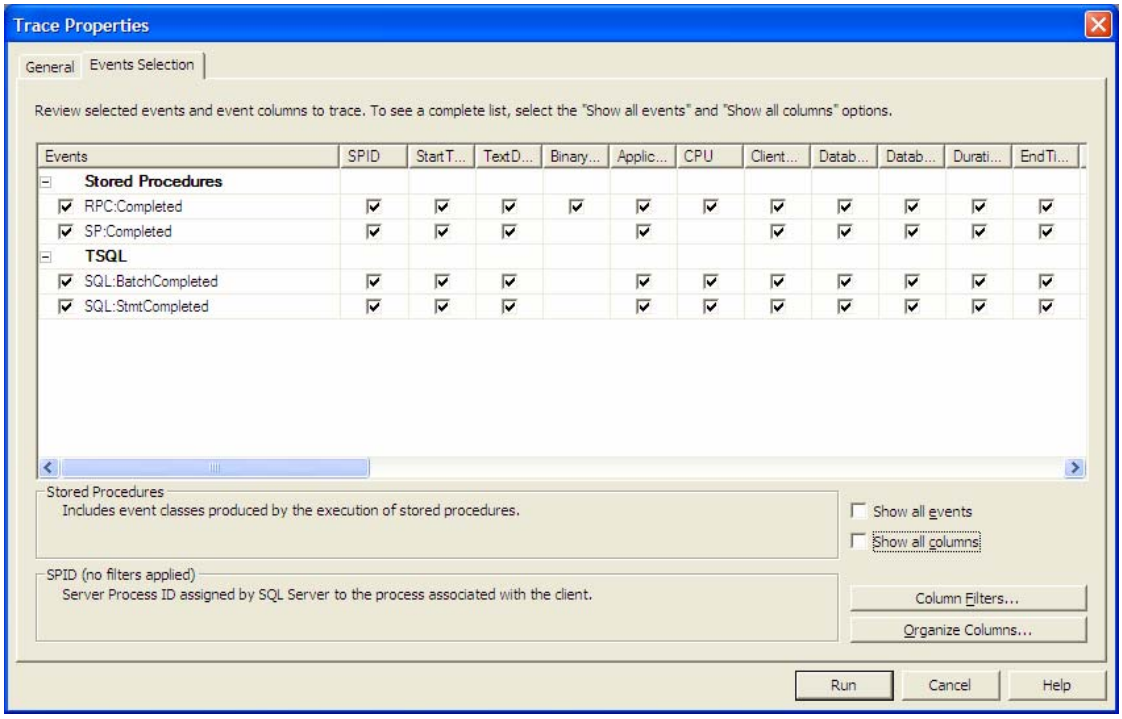


Figure 1.

Figure 2 shows the General tab in the same dialog. I have configured the trace to store into a table on a server other than the server I am tracing. I have also configured the trace to stop after an hour.

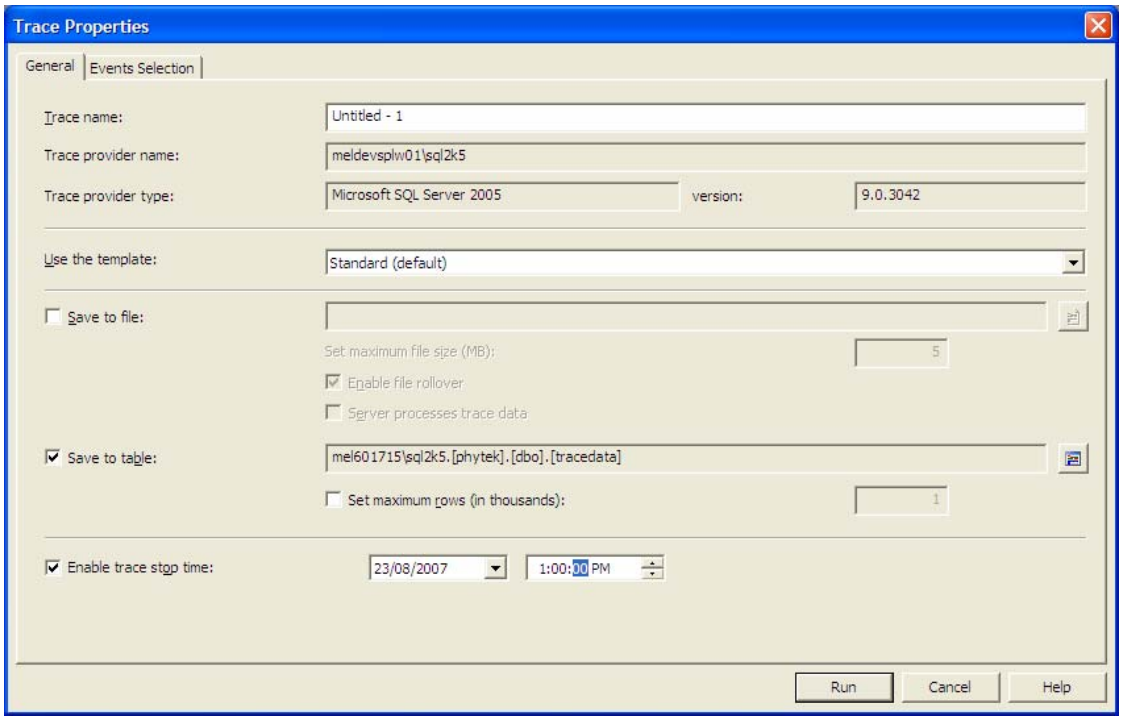


Figure 2.

Once the trace is finished, the data should now be available in the database table that I configured. For those who wish to use server side tracing, we will also assume from this point that the trace data now exists in a table.

On a server with a large amount of throughput, there will be a large number of rows in the trace table. In order to make sense of all this data, it will be necessary to aggregate. I suggest aggregating by at least the SQL text, or TextData column. You can include other columns in your aggregation, such as user or client host name, but for now I will concentrate on TextData.

TextData is a text column, which means I can't do a GROUP BY on it. So I will convert it to something we can do a GROUP BY on. In order to do this, I will create a column on the trace table called TextDataTrunc. Figure 3 illustrates the populating of this column with a simple UPDATE.

> To get a more accurate aggregation, it would be better to process the TextData column and replace the parameters and literals with some token that allows the SQL statements to be hashed. The hash could then be stored, and the aggregation could be performed on the hash value. This could be done with a C# user-defined function on SQL Server 2005. Illustrating how to do this is beyond the scope of this paper, so I am using the quick and dirty method.

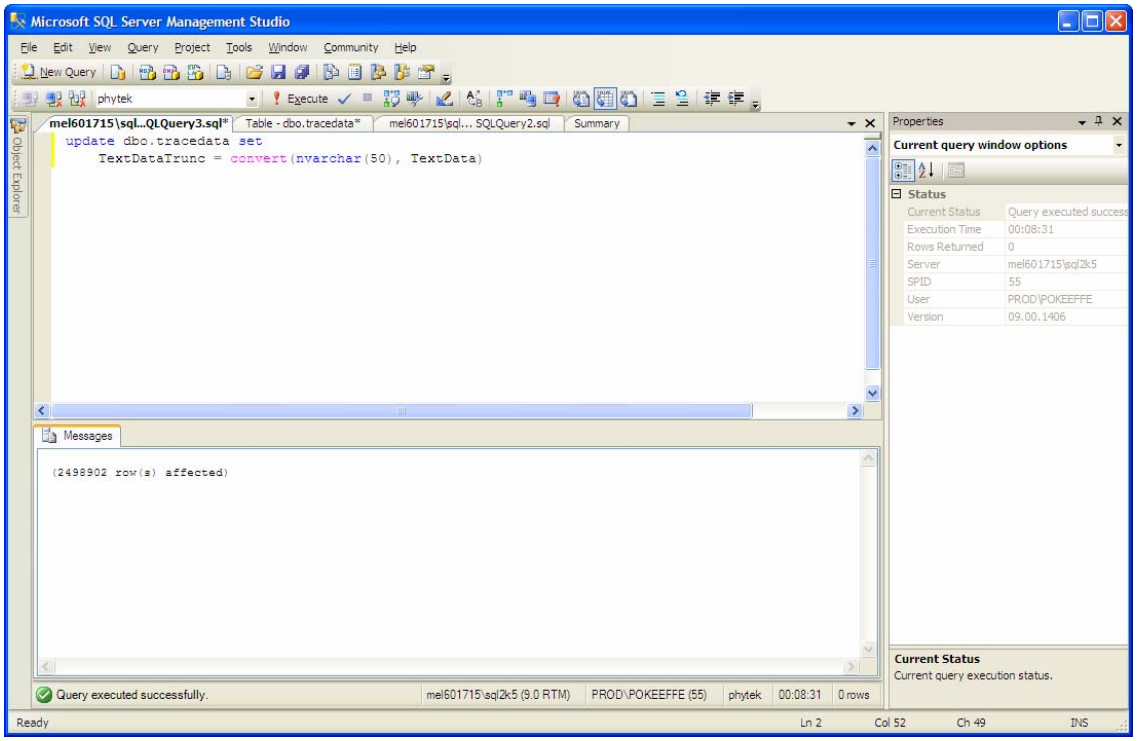


Figure 3.

Once the UPDATE is completed, I can now query the table to get the data we require.

For example, say you wanted to know the SQL that had been executed the most—I could use a simple query:

```
select
      TextDataTrunc,
      Count(TextDataTrunc) as ExecCount,
      Sum(cpu) as TotalCPU
      Avg(cpu) as AvgCPU
from
      dbo.tracedata
where
      EventClass = 12
group by
      TextDataTrunc
order by
      ExecCount desc
```

Figure 4 shows an example of this.

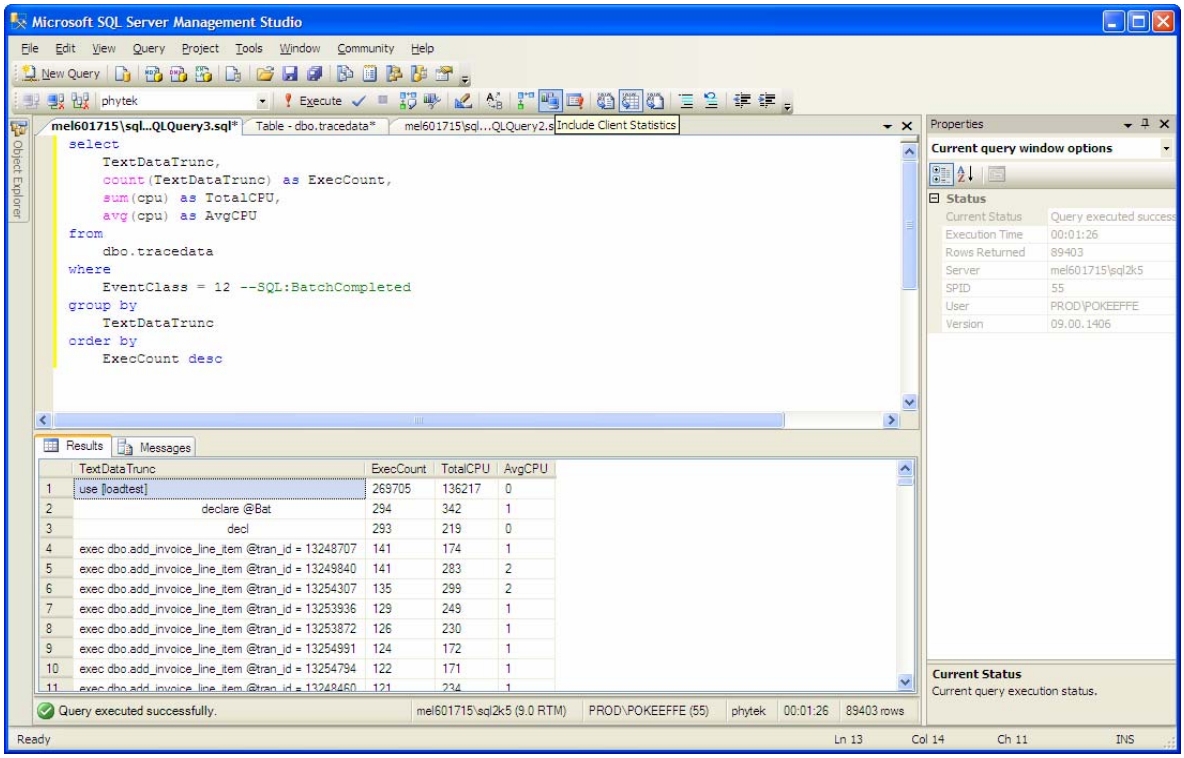


Figure 4

The values to use for the EventClass column can be found in SQL Server books online under the topic sp_trace_setevent.

5. ZEN AND THE ART OF NEGOTIATING WITH YOUR SAN ADMINISTRATOR

Storage area networks (SANs) are fantastic. They offer the ability to provision and manage storage in a simple and easy way.

Even though SANs can be configured for fast performance from a SQL Server perspective, they often aren't. Organizations usually implement SANs for reasons such as storage consolidation and ease of management, not for performance. To make matters worse, generally you do not have direct control over how the provisioning is done on a SAN. Thus, you will often find that the SAN has been configured for one logical volume where you have to put all the data files.

Having all the files on a single volume is generally not a good idea if you want the best I/O performance. As an alternative, you will want to:

- Place log files on their own volume, separate from data files. Log files are almost exclusively written and not read. So you would want to configure for fast write performance
- Place tempdb on its own volume. tempdb is used for myriad purposes by SQL Server internally, so having it on its own I/O subsystem will help

To further fine-tune performance, you will first need some stats. There are, of course, the Windows disk counters, which will give you a picture of what Windows thinks is happening (don't forget to adjust raw numbers based on RAID configuration). Also, SAN vendors often have their own performance data available. SQL Server also has file level I/O information available in the form of a function fn_virtualfilestats. From this function, you can:

- Derive I/O rates for both reads and writes
- Get I/O throughput
- Get average time per I/O
- Look at I/O wait times

Figure 5 shows the output of a query using this function ordered by IoStallMS, which is the amount of time users had to wait for I/O to complete on a file.

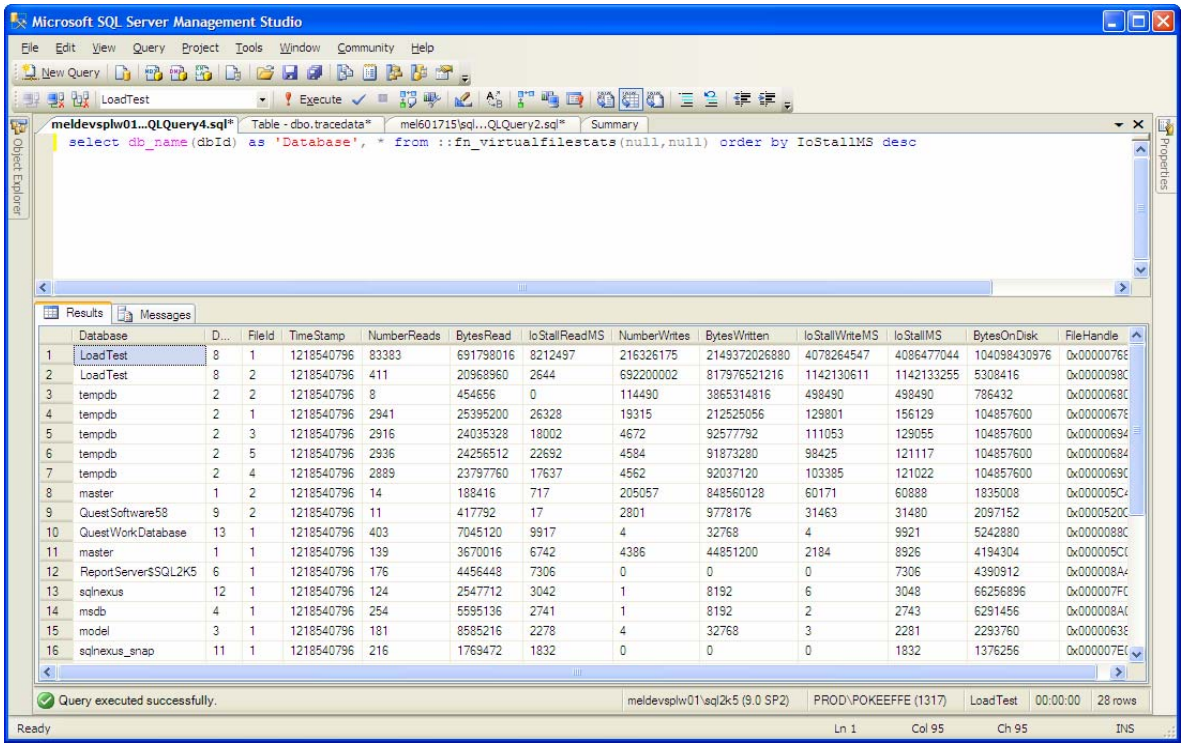


Figure 5.

Using these numbers, you can quickly narrow down which files are responsible for consuming I/O bandwidth and ask questions such as:

- Is this I/O necessary? Am I missing an index?
- Is it one table or index in a file that is responsible? Can I put this index or table in another file on another volume

4. THE HORROR OF CURSORS (AND OTHER BAD T-SQL)

There is a blog I read every day — www.thedailywtf.com (wtf stands for Worse Than Failure, of course). Readers post real experiences they had with bad organizations, processes, people and code. In it I found this gem:

```
DECLARE PatientConfirmRec CURSOR FOR
SELECT ConfirmFlag
FROM Patient where policyGUID = @PolicyGUID
OPEN PatientConfirmRec
FETCH NEXT FROM PatientConfirmRec
WHILE @@FETCH_STATUS = 0
BEGIN
UPDATE Patient
SET ConfirmFlag = 'N'
WHERE CURRENT OF PatientConfirmRec
FETCH NEXT FROM PatientConfirmRec
END
CLOSE PatientConfirmRec
DEALLOCATE PatientConfirmRec
```

This is real code in a real production system. It can actually be reduced to:

```
UPDATE Patient SET ConfirmFlag = 'N'
WHERE PolicyGUID = @PolicyGUID
```

This refactored code of course will run much more efficiently, allow the optimizer to work its magic and take far less CPU time. In addition, it will be far easier to maintain. It's important to schedule a code review of the T-SQL in your applications, both stored code and client side, and to try to refactor such nonsense.

Bad T-SQL can also appear as inefficient queries that do not use indexes, mostly because the index is incorrect or missing. It's important to learn how to tune queries using query plans in SQL Server Management Studio. Figure 6 shows an example of a large query plan.

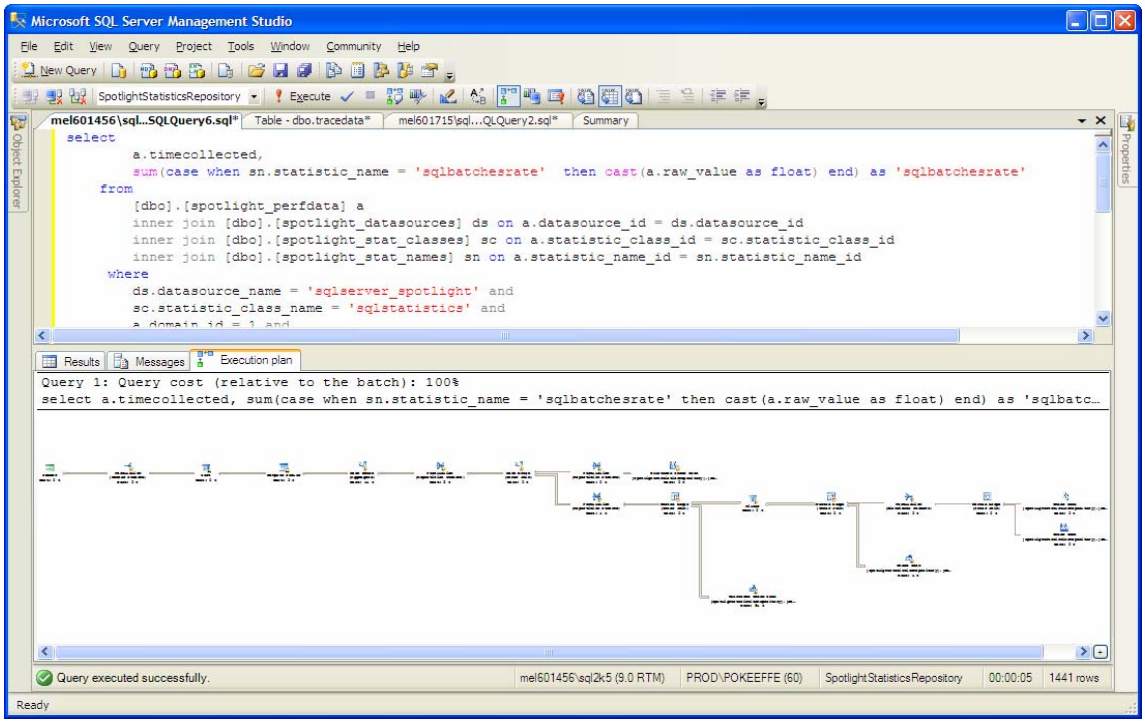


Figure 6

A detailed discussion of query tuning using query plans is beyond the scope of this white paper. However, the simplest way to start this process is by turning SCAN operations into SEEKs. SCANs will read every row in the table. For large tables, it is expensive in terms of I/O, whereas a SEEK will use an index to go straight to the required row. This, of course, requires an index to use, so if you find SCANs in your workload, you could be missing indexes.

There are a number of good books on this topic, including:

- *SQL Server Query Performance Tuning Distilled, Second Edition (paperback)* by Sajal Dam
- *Microsoft SQL Server 2005 Performance Optimization and Tuning Handbook (paperback)* by Ken England, Gavin JT Powell

3. PLAN REUSE – RECYCLING FOR SQL

Before executing a SQL statement, SQL Server first creates a query plan. This defines the method SQL Server will use to satisfy the query. Creating a query plan requires significant CPU. Thus, SQL Server will run more efficiently if it can reuse query plans instead of creating a new one each time a SQL statement is executed.

There are some performance counters available in the SQL Statistics performance object that will tell you whether you are getting good plan reuse.

```
(Batch Requests/sec – SQL Compilations/sec) / Batch Requests/sec
```

This formula tells you the ratio of batches submitted to compilations. You want this number to be as small as possible. A 1:1 ratio means that every batch submitted is being compiled, and there is no plan reuse at all.

It's not easy to pin down the exact workload that is responsible for poor plan reuse, because the problem usually lies in the client application code that is submitting queries.

You therefore may need to look at the client application code that is submitting queries. Is it using prepared parameterized statements?

Using parameterized queries not only improves plan reuse and compilation overhead, but it also reduces the SQL injection attack risk involved with passing parameters via string concatenation.

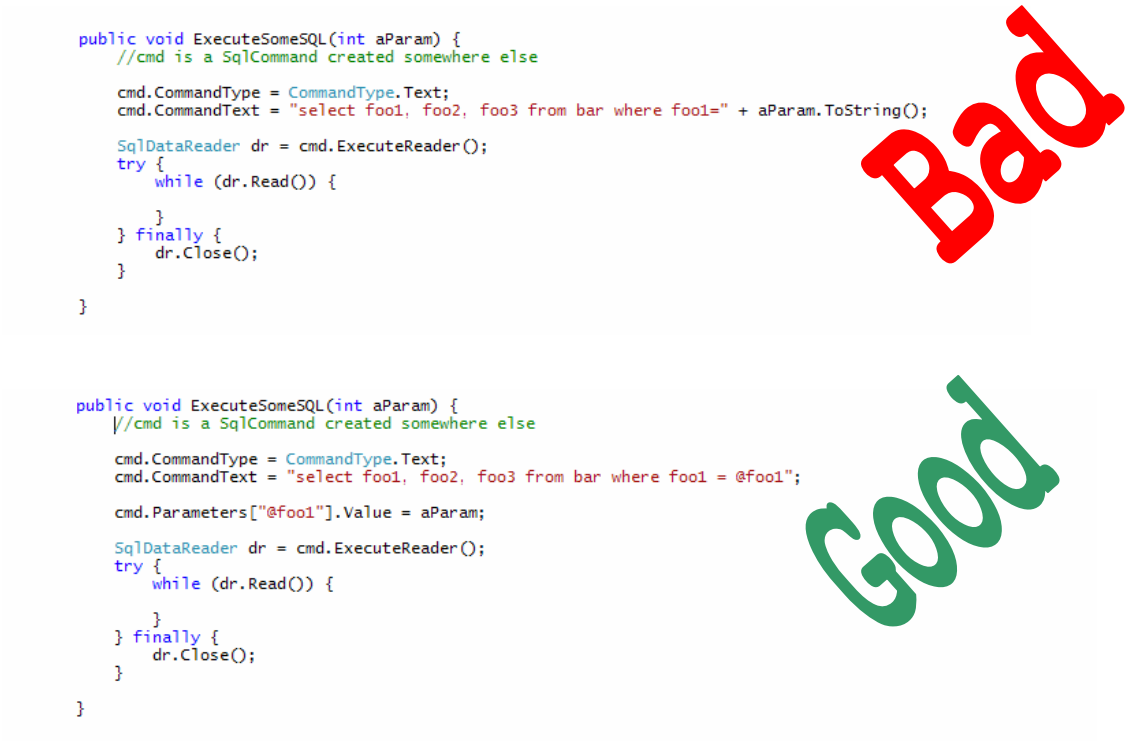


Figure 7

Figure 7 shows two code examples. Though they are contrived, they illustrate the difference between building a statement through string concatenation and using prepared statements with parameters.

SQL Server cannot reuse the plan from the ‘Bad’ example. If a parameter had been a string type, this function could be used to mount a SQL injection attack.

The ‘Good’ example is not susceptible to a SQL injection attack because a parameter is used, and SQL Server is able to reuse the plan.

2. THE MYSTERY OF THE BUFFER CACHE

The buffer cache is a large area of memory used by SQL Server to optimize physical I/O.

No SQL Server query execution reads data directly off the disk. The database pages are read from the buffer cache. If the sought-after page is not in the buffer cache, a physical I/O request is queued. Then the query waits and the page is fetched from the disk.

Changes made to data on a page from a DELETE or an UPDATE operation are also made to pages in the buffer cache. These changes are later flushed out to the disk.

This whole mechanism allows SQL Server to optimize physical I/O in several ways:

- Multiple pages can be read and written in one I/O operation
- Read ahead can be implemented. SQL Server may notice that for certain types of operations, it could be useful to read sequential pages—the assumption being that right after you read the page requested, you will want to read the adjacent page

There are two indicators of buffer cache health:

1. MSSQL$Instance:Buffer Manager\Buffer cache hit ratio – This is the ratio of pages found in cache to pages not found in cache. Thus, the pages need to be read off disk. Ideally, you want this number to be as high as possible. It is possible to have a high hit ratio but still experience cache thrashing.
2. MSSQL$Instance:Buffer Manager\Page Life Expectancy – This is the amount of time that SQL Server is keeping pages in the buffer cache before they are evicted. Microsoft says thata page life expectancy greater than five minutes is fine. If the life expectancy falls below this, it can be an indicator of memory pressure (not enough memory) or cache thrashing.

Cache thrashing is the term used when a large table or index scan is occurring. Every page in the scan must pass through the buffer cache. This is very inefficient because the cache is being used to hold pages that are not likely to be read again before they are evicted.

Since every page must pass through the cache, other pages need to be evicted to make room. A physical I/O cost is incurred because the page must be read off disk. Cache thrashing is usually an indication that large tables or indexes are being scanned.

To find out which tables and indexes are taking up the most space in the buffer cache, you can examine the cachesyscacheobjects on SQL Server 2000 or sys.dm_os_buffer_descriptors on SQL Server 2005.

The examplequerybelow illustrateshow to access the list of tables/indexes that are consuming space in the buffer cache on SQL Server 2005:

```
select
      o.name,
      i.name,
      bd.*
from
      sys.dm_os_buffer_descriptors bd
      inner join sys.allocation_units a on bd.allocation_unit_id =
a.allocation_unit_id
      inner join sys.partitions p      on (a.container_id = p.hobt_id and a.type in
(1,3)) or (a.container_id = p.partition_id and a.type = 2 )
      inner join sys.objects o on p.object_id = o.object_id
      inner join sys.indexes i on p.object_id = i.object_id and p.index_id =
i.index_id
```

You can also use the new index DMVs to find out which tables/indexes have large amounts of physical I/O.

1. THE TAO OF INDEXES

SQL Server 2005 gives us some very useful new data on indexes.

sys.dm_db_index_operational_stats

sys.dm_db_index_operational_stats contains information on current low-level I/O, locking, latching and access method activity for each index.

Use this DMV to answer the following questions:

- Do I have a 'hot' index? Do I have an index on which there is contention? – The row_lock_wait_in_ms/page_lock_wait_in_ms columns can tell us whether there have been waits on this index
- Do I have an index that is being used inefficiently? Which indexes are currently I/O bottlenecks? – The page_io_latch_wait_ms column can tell us whether there have been I/O waits while bringing index pages into the buffer cache – a good indicator that there is a scan access pattern
- What sort of access patterns are in use? The range_scan_count and singleton_lookup_count columns can tell us what sort of access patterns are used on a particular index

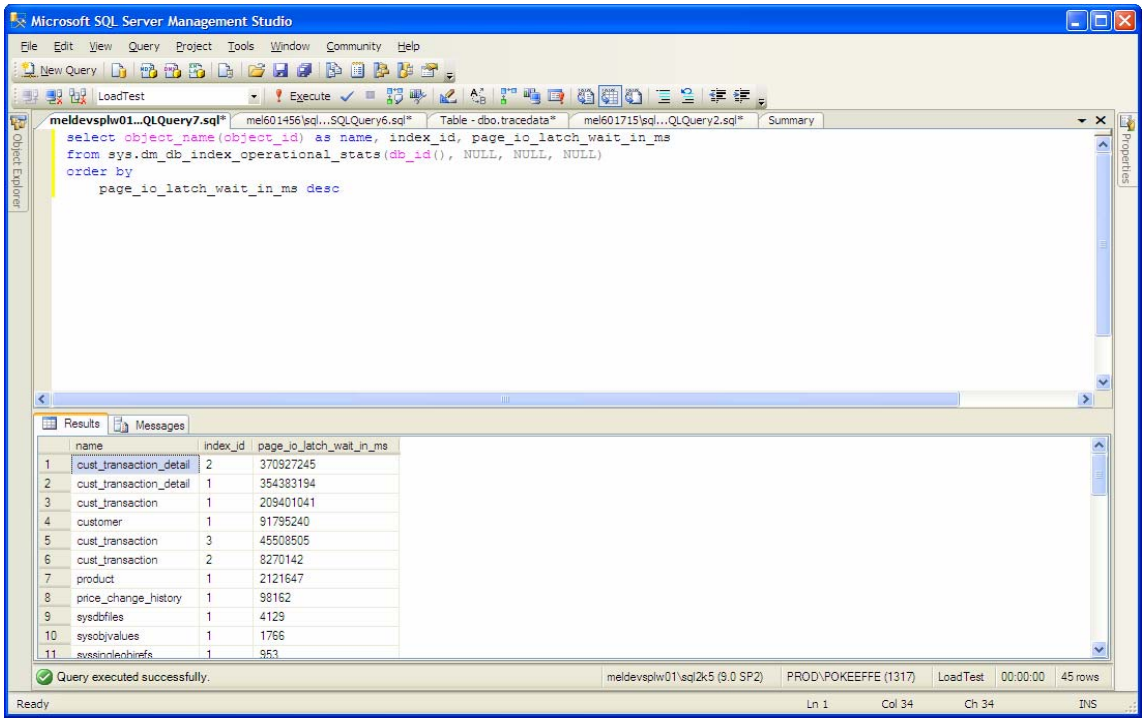


Figure 8.

Figure 8 illustrates the output of a query that lists indexes by the total PAGE_IO_LATCH wait. This is very useful when trying to determine which indexes are involved in I/O bottlenecks.

sys.dm_db_index_usage_stats

sys.dm_db_index_usage_stats contains counts of different types of index operations and the time each type of operation was last performed.

Use this DMV to answer the following questions:

- How are users using the indexes? The user_seeks, user_scans, user_lookups columns can tell you the types and significance of user operations against indexes
- What is the cost of an index? The user_updates column can tell you what the level of maintenance is for an index
- When was an index last used? The last_* columns can tell you the last time an operation occurred on an index

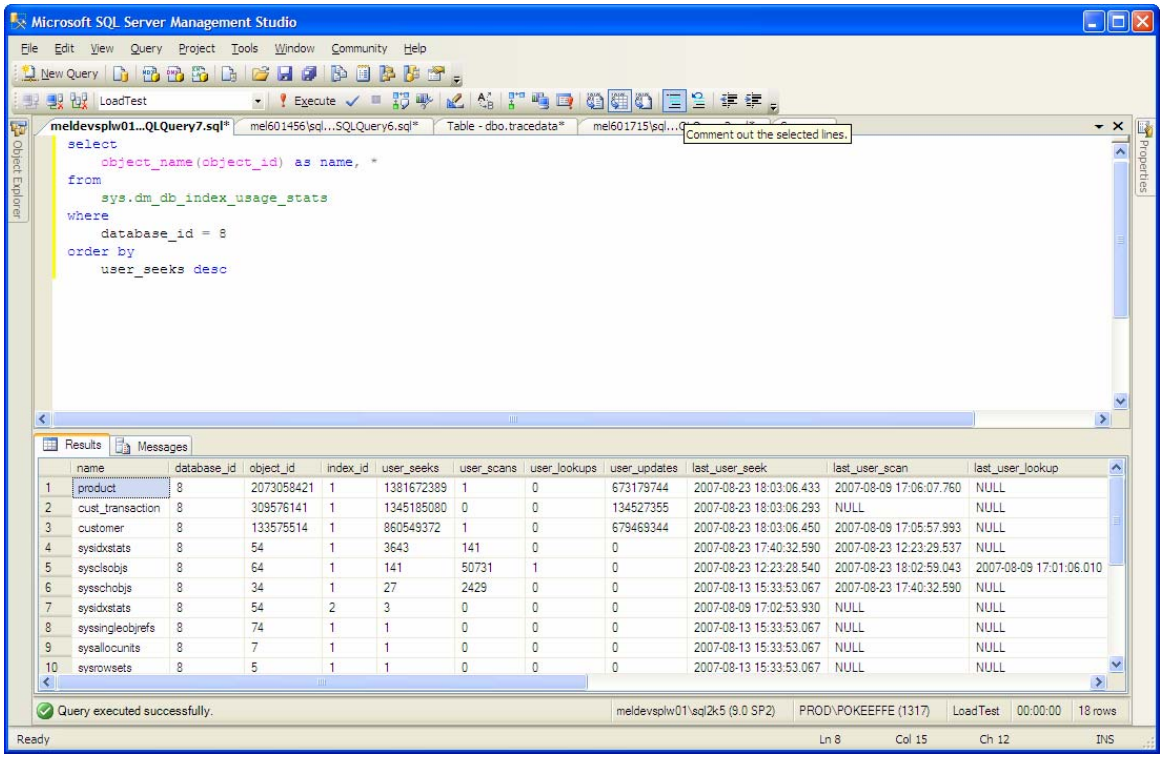


Figure 9.

Figure 9 illustrates the output of a query that lists indexes by the total number of user_seeks. If you instead wanted to identify indexes that had a high proportion of scans, you could order by the user_scans column.

Now that you have an index name, wouldn't it be good if you could find out what SQL statements used that index?

On SQL Server 2005 – now you can.

AND AN EXTRA ONE FOR GOOD MEASURE. LEARN XPATH

Say we had an index name received from a query on sys.dm_os_buffer_descriptors or from sys.dm_db_index_operational_stats. How do we find the SQL Statements that use that index?

Microsoft did two really cool things (amongst all the other really cool things) in SQL Server 2005:

1. Exposed the contents of the procedure (or plan) cache through sys.dm_exec_query_stats and more specifically the column containing the compiled plan in XML
2. Embedded the ability to use XPath expressions on XML columns

XPath is a way of querying an XML DOM (Document Object Model). What are the SQL Statements that use a particular index? An example is shown below.

```
select sql.text,
  substring(sql.text, statement_start_offset/2,
  (case when statement_end_offset = -1 then
  len(convert(nvarchar(max), text)) * 2
  else statement_end_offset end - statement_start_offset)/2),
  qs.execution_count,
  qs.*,
  p.*
from
  sys.dm_exec_query_stats as qs
  cross apply sys.dm_exec_sql_text(sql_handle) sql
  cross apply sys.dm_exec_query_plan(plan_handle) p
where
  query_plan.exist('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/showplan";
/ShowPlanXML/BatchSequence/Batch/Statements//Object/@Index[. =
"[IDX_cust_transaction_cust_id]"]') = 1
```

Here is another example of all the SQL Statements that do a clustered index scan.

```
declare @op nvarchar(30)
set @op = 'Clustered Index Scan'

select top 10
       sql.text,
       substring(sql.text, statement_start_offset/2,(case when statement_end_offset
= -1 then len(convert(nvarchar(max), text)) * 2 else statement_end_offset end -
statement_start_offset)/2),
       qs.execution_count, qs.*, p.*
from sys.dm_exec_query_stats AS qs
cross apply sys.dm_exec_sql_text(sql_handle) sql
cross apply sys.dm_exec_query_plan(plan_handle) p
where query_plan.exist('
declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/showplan";
/ShowPlanXML/BatchSequence/Batch/Statements//RelOp/@PhysicalOp[. =
sql:variable("@op")]
') = 1 and total_logical_reads/execution_count > 1000
order by total_logical_reads
```

There are many other ways you can use this mechanism.

For example, SQL Server 2005 plans have data that tells you that SQL Server would have used an index on a column had it been present. You can use this method to find these plans.

You can find queries that have large estimates for the numbers of rows being read to identify queries that read large numbers of rows.

You can identify lookup operations that could be optimized using a covering index. Since the columns to be returned are known, and the indexes that the plan used are known, you can add columns to the index so it can be used as a covering index.

CONCLUSION

On reflection, there are far more than ten (or eleven) things you should know about SQL Server performance. However, this white paper offers a good starting point and some practical tips about performance optimization that you can apply to your SQL Server environment.

ABOUT THE AUTHOR

Patrick O'Keeffe is a senior software architect at Quest Software, where he specializes in the design and implementation of diagnostic and performance tools for Microsoft SQL Server. Patrick has more than 12 years of experience in software engineering and architecture, and is based in Quest's Melbourne Office.

ABOUT QUEST SOFTWARE, INC.

Quest Software, Inc. delivers innovative products that help organizations get more performance and productivity from their applications, databases and Windows infrastructure. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 50,000 customers worldwide meet higher expectations for enterprise IT. Quest Software can be found in offices around the globe and at www.quest.com.

Contacting Quest Software

Phone:	949.754.8000 (United States and Canada)
Email:	info@quest.com
Mail:	Quest Software, Inc. World Headquarters 5 Polaris Way Aliso Viejo, CA 92656 USA
Web site	www.quest.com

Please refer to our Web site for regional and international office information.

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract. Quest Support provides around the clock coverage with SupportLink, our web self-service. Visit SupportLink at http://support.quest.com

From SupportLink, you can do the following:

- Quickly find thousands of solutions (Knowledgebase articles/documents).
- Download patches and upgrades.
- Seek help from a Support engineer.
- Log and update your case, and check its status.

View the ***Global Support Guide*** for a detailed explanation of support programs, online services, contact information, and policy and procedures. The guide is available at: http://support.quest.com/pdfs/Global Support Guide.pdf

NOTES

[1] When using Windows disk counters, it is necessary to adjust for RAID configurations. You can do this using the following formulas:

Raid 0	value = (reads + writes) / number of disks
Raid 1	value = [reads + (2 * writes)] / 2
Raid 5	value = [reads + (4 * writes)] / number of disks
Raid 10	value = [reads + (2 * writes)] / number of disks