QUEST SOFTWARE®

# Making Sense of Your SQL Server Application's Performance – A Practical Guide

*Written by,*
*Patrick O'Keeffe*
*Quest Software, Inc.*

World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
e-mail: info@quest.com
U.S. and Canada: 949.754.8000

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

Updated—May 1, 2007

# CONTENTS

# INTRODUCTION

Performance tuning on SQL Server can be a daunting task. The volume of tuning data to access on various Web sites - such as which counters to look at, which queries to run — can be overwhelming.

One issue that is often overlooked is the how the SQL that is executed on the server affects performance, and how to use that understanding to tune performance more effectively.

In this white paper, a simple framework for gaining this understanding is presented along with some insight and advice for getting the most performance from your SQL Server application.

# WHAT CHALLENGE ARE WE TRYING TO SURMOUNT?

The challenge is a simple one – DBAs try to get the most value they can from their SQL Server deployments. They ask questions such as, "am I getting the best efficiency? Or, "will my application scale?"

A scalable system is one in which the demands on the database server increase in a predicable and reasonable manner. For instance, doubling the transaction rate might double the demand on the database server, but a quadrupling of demand could easily result in a system failure.

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 the hardware investment, a DBA needs to ensure that the SQL or application workload running on the database servers is executing as quickly and as efficiently as possible.

A SQL Server at idle has no performance problems; it is only when it executes a workload – your application – that problems arise. Therefore, you can optimize the performance of your SQL Server by understanding how your application workload behaves.

There are a number of drivers for performance tuning:

- Meeting service level agreement (SLA) targets
- Improving efficiency to free up resources for other purposes
- Ensuring scalability to maintain SLAs into the future

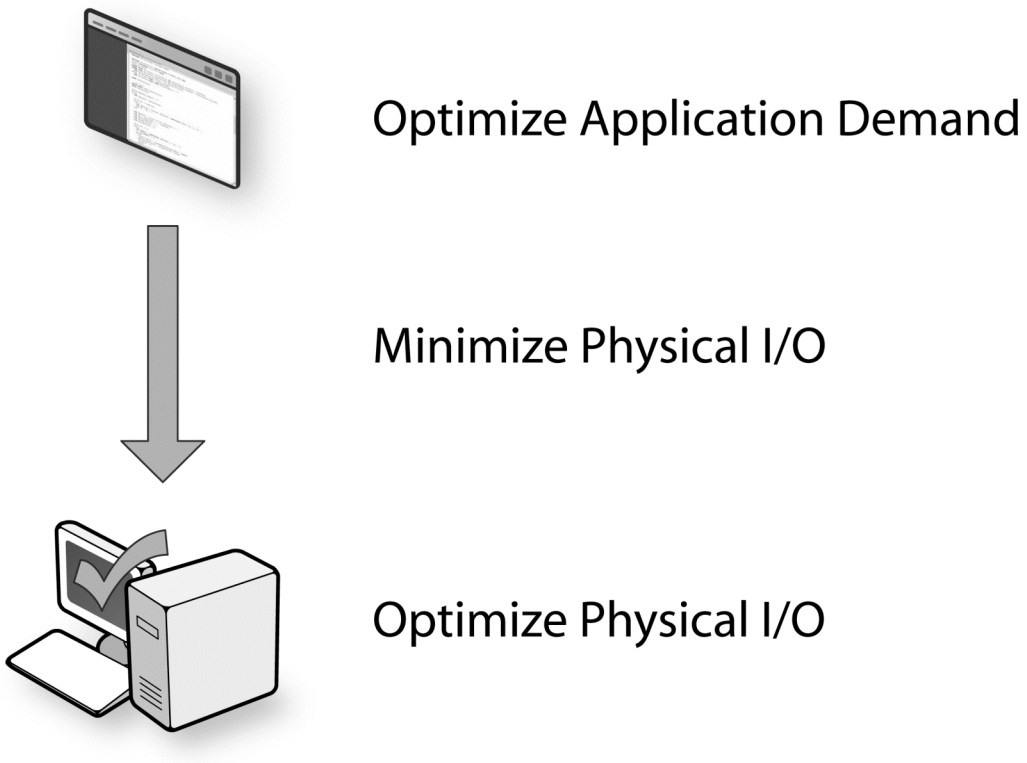
Performance optimization is typically an ongoing process. For SLA targets, you might consider tuning complete if you meet SLAs. But for the other two drivers above, you're never really finished with tuning and should continued doing it until the performance is good enough. When the performance of the application ceases to be good enough, you should resume tuning.

Performance that is good enough is usually defined by business imperatives such as SLAs or system throughput requirements. Beyond these requirements, it's wise to maximize the scalability and efficiency of all database servers.

The performance optimization approach described in this paper is a simple three-step process. You should perform it in pre-production or staging, as well as in production – periodically, or as required.

# THREE FOR THREE (OR WHERE DO I START?)

To optimize performance, you need a simple set of steps that will help you understand how your application workload is behaving. From here, you can make necessary changes to its behavior.




Let's say we determine an I/O bottleneck as the root cause of a performance problem. We add more disk spindles (i.e. disk devices) and the problem goes away for a while.

The problem with this approach is that we jumped to the last step, *optimize physical I/O,* and temporarily concealed the problem. We started at the hardware end, without first looking at the application end.

In short, we didn't *optimize application demand* — We never asked questions such as:

- Is the bottleneck due to a large number of reads?
- Are a large number of reads necessary for my application to meet its requirements?
- Do we have non-optimal SQL?

- Did a developer forget a WHERE clause?
- Do we have a non-optimal physical model?
- Are we missing an index?

We *didn't minimize the physical I/O* or ask:

- Would adding more memory to the server reduce the physical IO rate?
- Would adding more memory cost less than adding more disks?

As you can see, there are a numbers of issues to investigate before we purchase new hardware. Only when we finish the first two steps should we be investigating file placement on disk and considering more or faster I/O bandwidth (spindles).

For each of the steps above, we can perform three tasks:

- Identify bottleneck
- Find the workload that is causing the bottleneck
- Fix the bottleneck

**Identifying Bottlenecks**

You know you have a bottleneck if you are monitoring your application for it in production, you get alerted to it, or your users are calling!

If you have not yet deployed your application to production, you need to run a simulation. You can achieve this by:

- observing the application in a real-time test environment
- playing back a recording of the application executing real-time
- simulating the bottleneck

**When your users are calling, a bottleneck situation is generally much worse than it would be if you were monitoring for it. Save yourself some trouble and find a solution to get early warnings**

You will get the best outcome by observing the actual workload in real-time or a recording of it. Ideally, you would also want to run the workload on hardware comparable to that which 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.

Once the workload is running, you need to observe its behaviour. You are looking for bottlenecks such as:

- Queries that are:
  - falling outside an established or ad-hoc performance criteria and showing signs of inefficient or unscaleable access paths (e.g., unnecessary or undesirable full table scans),
  - consuming an unreasonable share of system resources (thus presenting opportunities for tuning)

- Contention within the database server, (e.g., lock contention)
- Unnecessary IO that might be reduced through more effective memory management
- Hardware resource:
    - CPU pressure
    - IO pressure
    - Memory pressure

More detail on types of bottlenecks and how to detect them will be covered later in this document.

### Find the Workload that is Causing the Bottleneck

The steps to identify workloads will vary widely.

- For a long-running query, the workload **is** the query
- For lock contention - blocking is usually a symptom, thus analysis of the blocking tree should reveal the cause
- For general resource (i.e., CPU, IO) bottlenecks - workload identification is covered later in this document

### Fix the Bottleneck and Start Again

There are various causes of performance problems that are often related to application design issues. Solutions for major problems are beyond the scope of this white paper. However, if the problem is general, there's a general solution that will be provided.

A good practice when you make changes based on your observations is to pick one problem to solve at a time. There are a few of reasons for this. If you fix a lot of problems at once and throughput improves, you won't know which fixes made the difference. Conversely, if throughput decreases, you won't know which change made the problems worse. Also, the database workload is highly interrelated, so a change to fix one problem may cause another problem, either directly or indirectly.

# THE PRACTICAL GUIDE

There is plenty of prescriptive SQL Server tuning advice that goes something like this:

"If counter x says y, then it means you have memory pressure so you should consider adding more memory."

While this may be true, it misses the point – counters alone do not tell the whole story, and though they are useful, they are a means to an end. It is all about making sure that your workload is executing as efficiently as you can make it. It is not about making sure that counter value z stays under 100.

This paper is about making sense of your application workload, understanding how it executes and what effect it has on your server. We will examine some typical resource bottlenecks and explain how to identify the workloads that cause them.

There are three types of bottlenecks that we will discuss:

- CPU
- IO Memory
- Blocking

There are other bottlenecks that we could discuss (e.g., tempdb), but they are more granular variations of the three major ones listed above. In any case, the goal is to apply the same steps to any bottleneck to determine the workload.

## *Do I Have a CPU Bottleneck?*

The most direct way to determine whether the SQL Server has a CPU bottleneck is to look at the Processor\% Processor Time performance counters for sustained high usage.

Don't forget to consider multiple CPUs and any CPU affinity you have configured. It is possible to have a CPU pressure condition even when all CPUs are not fully utilized. Another indicator of CPU pressure is when a significant proportion -- more than 25% -- of total wait time is signal wait.

## Wait? What's a Wait?

In a multi-threaded server, the SQL Server data flows from one subsystem to another, and hardware resources like CPU, memory and disk, are shared.

The shared worker threads must be queued to gain access to these resources. When a worker thread executing a task (SQL that has been submitted to the server) can't access a resource immediately, it enters a wait state where it will stay until the resource is available.

Signal Wait is a special type of wait that occurs when a worker is granted access to a resource but the worker still needs to wait for CPU time. Thus, Signal Wait indicates that workers are queuing up, waiting for CPU time.

## Finding the Top Five Wait Types

It is useful to know the top five wait types within your SQL Server. Particular wait types may indicate embryonic conditions that may turn into crises later on.

### SQL Server 2000

```
create table #waitstats (
        waittype varchar(80),
        requests numeric(20,1),
        waittime numeric (20,1),
        signalwaittime numeric(20,1))

insert into
        #waitstats (waittype, requests, waittime,signalwaittime)
exec
        ('dbcc sqlperf(waitstats)')

declare @totalwait numeric(20,1),
        @totalsignalwait numeric(20,1) ,
        @endtime datetime,
        @begintime datetime

--get the totals
select
        @totalwait=waittime, @totalsignalwait=signalwaittime
from
        #waitstats where [waittype] = 'Total'

--- subtract waitfor, sleep, and resource_queue from Total
select
        @totalwait = @totalwait - sum(waittime), @totalsignalwait = @totalsignalwait -
sum(signalwaittime)
from
        #waitstats
where
        waittype in ('Waitfor','Sleep','Resource_Queue')

-- insert adjusted totals, rank by percentage descending
insert into #waitstats select '***total***', 0, @totalwait, @totalsignalwait

select
        waittype,
        waittime,
        percentage = cast(100 * waittime / @totalwait as numeric(20,1)),
        signalwaittime,
        percentagesw = cast(100 * signalwaittime / @totalsignalwait as numeric(20,1))
from
        #waitstats
where
        waittype not in ('waitfor','sleep','resource_queue','total')
order by percentage desc
```

```
select '%signal'= 100 * (@totalsignalwait / @totalwait)
drop table #waitstats
```

## SQL Server 2005

```
create table #waitstats (
	waittype varchar(80),
	requests numeric(20,1),
	waittime numeric (20,1),
	signalwaittime numeric(20,1))

insert into
	#waitstats (waittype, requests, waittime, signalwaittime)
select
	wait_type, waiting_tasks_count, wait_time_ms, signal_wait_time_ms
from
	sys.dm_os_wait_stats


declare @totalwait numeric(20,1),
	@totalsignalwait numeric(20,1) ,
	@endtime datetime,
	@begintime datetime

--get the totals
select
	@totalwait=sum(waittime), @totalsignalwait=sum(signalwaittime)
from
	#waitstats



--- subtract waitfor, sleep, and resource_queue from Total
select
	@totalwait = @totalwait - sum(waittime), @totalsignalwait = @totalsignalwait -
sum(signalwaittime)
from
	#waitstats
where
	waittype in ('Waitfor','Sleep','Resource_Queue')


-- insert adjusted totals, rank by percentage descending
insert into #waitstats select '***total***', 0, @totalwait, @totalsignalwait

select
	waittype,
	waittime,
	percentage = cast(100 * waittime / @totalwait as numeric(20,1)),
	signalwaittime,
	percentagesw = cast(100 * signalwaittime / @totalsignalwait as numeric(20,1))
from
	#waitstats
where
	waittype not in ('waitfor','sleep','resource_queue','total')
order by percentage desc

select '%signal'= 100 * (@totalsignalwait / @totalwait)
drop table #waitstats
```

Access SQL Server books online, the MSDN library (msdn.microsoft.com) and other sources to obtain detailed description on columns.

It is important to note that the two queries above will return information since the time the server had been started or the counters had reset.

## *Finding the Top Five CPU Consumers*

Assuming you encounter a CPU bottleneck, you would first want to find out who is using all of the CPU resources. SQL Server uses CPU to execute queries, so let's examine the top five CPU consumers on the system:

### SQL Server 2000

To obtain this information on SQL Server 2000, you have to either use a trace or take a delta on the sysprocesses table. Aggregating trace data is rather inconvenient unless you have the data at hand, thus a delta is a quick and easy way to get a pointer in the right direction. A delta measures the difference between the current and previous collection which makes it simple to see which spid (user session) used CPU and how much it used during the collection period. A collection period of between 15 and 30 seconds is useful.

Note that you will need to use fn_getsql or DBCC INPUTBUFFER to get the SQL of the spids (information on how to do this is available in SQL Server books online).

```
if not exists (select 1 from dbo.sysobjects where name = 'sysprocesess_delta')
        create table sysprocesess_delta (
                spid int,
                cpu int,
                physical_io int,
                memusage int)

--compare last reading to current reading
select top 5
        s.spid,
        p.cpu - s.cpu delta_cpu,
        p.physical_io - s.physical_io delta_io,
        p.memusage - s.memusage delta_mem,
        p.*
from
        sysprocesess_delta s full outer join master.dbo.sysprocesses p on s.spid = p.spid
where
        p.cpu - s.cpu <> 0
order by
        delta_cpu desc

--delete the current old values
delete from sysprocesess_delta
--insert the current values
insert into
        sysprocesess_delta
select
        spid, cpu, physical_io, memusage from
master.dbo.sysprocesses
```

### SQL Server 2005

Getting this sort of information is easy – a simple DMV query gives us:

```
--Top CPU Consumers

select top (5)
        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
```

## *Finding Workload with Low Plan Re-use*

Before executing a SQL statement, SQL Server first creates a *query plan* – which defines the method SQL Server uses to satisfy the query. Creating a query plan takes significant CPU. SQL Server will run more efficiently if it can re-use query plans instead of creating a new one each time a SQL statement is executed. Whether or not plans are re-used depends on how SQL is submitted to the server.

There are some performance counters available in the SQL statistics performance object that will tell you if we are getting good plan re-use.

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

The formula above tells us 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 re-use at all.

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

You may need to look at the client application code that is submitting queries. Is it using prepared parameterized statements? Parameterized queries not only improve plan re-use and decrease compilation overhead, but they also reduce the SQL injection attack risk involved with passing parameters via string concatenation.

## *Do I Have Lock or Other Contention?*

Worker threads can wait on application resources, such as locks, just as they do on hardware resources. When this type of wait occurs, a spid is said to be blocked. Blocking can also be called contention (i.e., workers contend for shared resources).

There are some patterns to look for when investigating blocking:

- Single long wait
- Large number of waits on a single resource – serialization or 'hotspot'
- Large numbers of waits on large numbers of resources
- All of the above

In fact, when investigating the blocking pattern, it is likely to be the second and third case where a hierarchy of blocking occurs. Each spid will be blocking one another, thus it creates a tree.

## *Getting the Blocking Tree*

### SQL Server 2000

```
declare @qs_blocking_list table (
        spid int,
        blocked int,
        loginame sysname,
        nt_username sysname,
        lastwaittype sysname,
        waitresource sysname,
        status sysname,
        waittime bigint,
        program_name sysname,
        cmd sysname,
        cpu bigint,
        physical_io bigint,
        hostname sysname,
        dbid int
)

insert into @qs_blocking_list (
        spid ,
        blocked ,
        loginame,
        nt_username ,
        lastwaittype ,
        waitresource ,
        status ,
        waittime ,
        program_name ,
        cmd ,
```

```
        cpu ,
        physical_io ,
        hostname ,
        dbid
)
select
        spid ,
        blocked ,
        loginame,
        nt_username ,
        lastwaittype ,
        waitresource ,
        status ,
        waittime ,
        program_name ,
        cmd ,
        cpu ,
        physical_io ,
        hostname ,
        dbid
from master.dbo.sysprocesses

set nocount off

select  spid                                    as 'spid'
        ,       blocked                                 as 'BlockedBySPID'
        ,       rtrim(loginame)                 as 'SQLUser'
        ,       rtrim(nt_username)              as 'NTUser'
        ,       rtrim(lastwaittype)             as 'Type'
        ,   rtrim(waitresource)         as 'Resource'
        ,       rtrim(status) +
                        case when blocked > 0 then ' and blocked' else '' end +
                        case when spid in (select blocked from @qs_blocking_list) then ' and blocking'
else '' end as 'Status'
        ,       waittime                        as 'WaitTimeMS'
        ,       rtrim(program_name)             as 'Program'
        ,       rtrim(cmd)                              as 'Command'
        ,   cpu                                         as 'CPU'
        ,   physical_io                         as 'PhysicalIO'
        ,       rtrim(hostname)                 as 'HostName'
        ,       case
                        when dbid = 0 then ''
                else
                        db_name(dbid)
                end                                                     as 'dbid'
from @qs_blocking_list where blocked <> 0
or (spid in (select blocked from @qs_blocking_list))
```

## SQL Server 2005

```
select
        substring(st.text, (r.statement_start_offset/2)+1
                , ((case r.statement_end_offset
                     when -1 then datalength(st.text)
                     else r.statement_end_offset
                   end - r.statement_start_offset)/2) + 1) as statement_text,
```

```
        wt.*
from
        sys.dm_os_waiting_tasks wt
        inner join sys.dm_exec_requests r on wt.session_id = r.session_id
        cross apply sys.dm_exec_sql_text(r.sql_handle) as st
where
        wt.wait_type not in (
                'OLEDB',
                'LOGMGR_QUEUE',
                'LAZYWRITER_SLEEP',
                'REQUEST_FOR_DEADLOCK_SEARCH',
                'KSOURCE_WAKEUP',
                'BROKER_TRANSMITTER',
                'BROKER_EVENTHANDLER',
                'CHECKPOINT_QUEUE',
                'ONDEMAND_TASK_QUEUE',
                'SQLTRACE_BUFFER_FLUSH')
order by wt.wait_duration_ms desc
```

**Two important questions arise:**

What resource is in contention?

What SQL is each of the parties (contending for that resource) executing?

In SQL Server 2000, getting the SQL involved requires an extra step that includes fn_getsql or DBCC INPUTBUFFER.

Once you get this information, you can start at the root of the tree and ask yourself some more questions:

- What is being waited on? Can I eliminate the wait?
  - Let's say the contention is on a clustered index due to readers and writers. The answer might be that the readers should select fewer columns. This might allow the use of a covering index instead of the readers having to go to the base table.

- Can I shorten the wait? Do I have a client-side implementation that is holding a transaction open waiting for user input?

## Do I Have a Memory Bottleneck?

Generally, SQL Server will determine the best use of available memory, and you will normally allocate the majority of server memory to SQL Server. The following suggestions may help:

- Avoid locating multiple instances on the same host because this can complicate the memory allocation.
- Make sure that multiple instances are configured with a *max server memory* setting that limits memory usage so that the instances do not

fight each other over memory. Add more memory before adding disks. A small amount of memory can often help avoid a large amount of IO. Although adding more disks can prevent a disk bottleneck, this can never be as beneficial as avoiding the disk IO in the first place.

Let's assume that SQL Server is not starved of physical memory (i.e., there is no swapping) and the server is re-using query plans. If this is the case, you should turn your attention to how the buffer cache is behaving.

The best indicator of the buffer cache behavior is the Buffer Manager\Page Life Expectancy performance counter. This measures in seconds how long a page might remain in the buffer cache before being evicted to make room for other pages. A low number means that pages do not spend a long time in the cache, and cache thrashing is occurring.

Ideally, you want pages to stay in cache as long as possible. Every page found in cache is one less page that has to be read off the disk to satisfy a query.

Cache thrashing is most commonly due to a query reading a large number of rows, and it could indicate scan access patterns. A large table scan can flood the cache with pages that will never be re-read at the expense of pages that are frequently accessed.

You can find the largest objects in the buffer cache by querying sys.dm_os_buffer_descriptors on SQL Server 2005 and syscacheobjects on SQL Server 2000. Once you have a table and/or index name, look for workload that uses those tables or indexes.

## *Do I Have an IO Bottleneck?*

The most obvious way to determine whether the SQL Server has an IO bottleneck is to look at the disk performance counters. The two counters that indicate IO pressure are the Avg Disk Queue Length and the Avg Disk Sec/Read. Don't forget to consider RAID configurations when interpreting these counter values.

Another indicator of IO pressure is when waits on PAGEIOLATCH_* wait types account for a significant proportion of the total wait time These waits occur when a worker had to wait for a page to be read off the disk and into the buffer cache.

## *Finding the Top Five IO Consumers*

If you do have an IO bottleneck, you would first find out who is using all the IO bandwidth. Queries consume IO bandwidth when executing, so determine the top five IO consumers on the system:

### SQL Server 2000

Use a trace to get this information. Store the trace results to a table, or use a server side trace and then query.

### SQL Server 2005

On SQL Server 2005 we can simple issue a DMV query:

```
--Top IO Consumers

select top (5)
        (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
```

In both cases, look for a workload that produces a large number of logical or physical reads. Then, ask these questions:

- Why does the application execute a query that does large numbers of reads?
- Is this required?

In SQL Server 2005, we can see physical IO numbers. This means that to satisfy the query, SQL Server had to go to disk to get a page – a sign of scan access patterns.

## *Finding the Top Five SQL with Scan Access Patterns*

On SQL Server 2005, you can identify workload through:

- An aggregated representation of the plan cache including the actual plan and the SQL – sys.dm_exec_query_stats
- XPath queries

The following is an example of how both ways can be used to identify workload that uses a scan access pattern:

```
--Show me all SQL that does a Clustered Index Scan
declare @op nvarchar(30)
set @op = 'Clustered Index Scan'

select top (5)
	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
```

You can apply this in many ways. If you wanted to look for queries with plans that contain Bookmark Lookup access patterns, you could use a query similar to the one above.

## *Finding the Top Five SQL with Large Estimated Rows Read*

Finding queries that are reading large numbers of rows is useful when you are trying to reduce logical IO. This is another example of how you can leverage XPath to find the information you need:

```
--Show me all SQL that have ops that potentially read large numbers of rows
select top(5)
	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/@EstimateRows[. > 10000]') = 1
```

## *Finding the Top Five Indexes That Required IO Wait*

If wait statistics indicate that there are PAGEIOLATCH_* waits, you can query a DMV on SQL Server 2005 to find indexes that were involved in IO wait.

```
select top(5)
	object_name(ios.object_id) as table_name,
	si.name as index_name,
	ios.object_id,
	ios.index_id,
	ios.partition_number,
	ios.page_io_latch_wait_count,
	ios.page_io_latch_wait_in_ms
from
	sys.dm_db_index_operational_stats(db_id('LoadTest'), NULL, NULL, NULL) ios
	inner join sys.indexes si on si.object_id = ios.object_id and si.index_id = ios.index_id
where
	page_io_latch_wait_in_ms > 0
order by
	page_io_latch_wait_in_ms desc
```

## *Finding the Top 5 SQL That Use a Particular Index*

Assuming that we have significant PAGEIOLATCH_* wait and we have identified which indexes are involved in the wait, how do we find the SQL that is using those indexes?

Following is another good example of leveraging XPath queries:

```
--Show me all SQL that use a particular index
select top (5)
	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
```

# CONCLUSION

There are a number of key points that you should take away from this whitepaper:

- Asking questions to get a picture of how your application behaves is key to performance tuning.
- You should follow the performance optimization plan
    - Optimize application demand
    - Minimize logical I/O
    - Optimize physical I/O
    - For each:
        - Find the bottleneck
        - Identify the workload responsible
        - Fix the bottleneck
- You should leverage the new features in SQL Server 2005 to your advantage (XPath, DMVs)

Lastly, remember that a SQL Server at idle has no performance problems. Performance issues arise only when it executes a workload – your application. Always try to identify the SQL that is the root cause of an identified bottleneck.

# About the Author

Patrick O'Keeffe is a senior 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's Windows Management solutions simplify, automate and secure Active Directory, Exchange and Windows, as well as integrate Unix and Linux into the managed environment. 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