SQL Server Performance Tuning Using Wait Statistics: A Beginner's Guide

By Jonathan Kehayias
and Erin Stellato

SQLskills immerse yourself in sql server
simple talk

Content

Introduction

When a user application submits to SQL Server a request for data, the biggest element of SQL Server's total response time would, ideally, be the **CPU processing time**. In other words, the time it takes the CPU to pull together the required data, slice, dice, and sort it according to the query specifications and send it back. However, in a busy database system, with hundreds or thousands of user requests competing for the limited resources of the database server, there will be times when a request is **waiting** to proceed, rather than actively processing. For example, Session A's request may be waiting for Session B to release a lock on a resource to which it needs access.

Every time a request is forced to wait, SQL Server records the **length of the wait**, and the **cause of the wait**, a.k.a. the wait type, which generally indicates the **resource** on which the request was waiting. These are the **wait statistics**, and SQL Server exposes them primarily through two Dynamic Management Views:

- **`sys.dm_os_wait_stats`** (or `sys.dm_db_wait_stats` on Windows Azure SQL Database) – aggregated wait statistics for all wait types
- **`sys.dm_os_waiting_tasks`** – wait statistics for currently-executing requests that are experiencing resource waits

The basis of performance tuning SQL Server using wait statistics is simply that we interrogate these statistics to find out the primary reasons why requests are being forced to wait, and focus our tuning efforts on relieving those bottlenecks.

The SQLOS scheduler and thread scheduling

In order to understand wait statistics in SQL Server, and how SQL Server measures wait times, we need to delve a little into SQL Server's scheduling model.

> ***More on SQLOS***
> *When you're ready to dig deeper, Slava Oks's, A new platform layer in SQL Server 2005, is old but still serves as a good primer on SQLOS architecture. I also recommend highly Paul Randal's SQL Server Performance Troubleshooting Using Wait Statistics training course, (see the Further reading section at the end, for more details).*

When a user application requests a connection, SQL Server will authenticate the user and then create for that connection a **session**, identified by a unique **session_id**. Within the context of this connection and the associated session, the user application can submit **requests**, such as queries to return data. As soon as the user submits a query, SQL Server gets to work executing it.

SQL Server uses operating system threads, or **worker threads**, to perform the tasks necessary to complete a given process. Some threads are dedicated (for example, there is a dedicated thread to perform the `CHECKPOINT` process) but SQL Server maintains the others in a thread pool and uses them to process user requests.

SQL Server does its own thread scheduling via a pseudo-operating system called the SQLOS, which runs as a part of the SQL Server database engine, and also provides memory management and other functions to SQL Server.

When SQL Server receives a user request, a SQLOS scheduler assigns it to a 'parent' task, and the task to a worker thread. If SQL Server decides to parallelize a task, the assigned parent task remains and the SQLOS creates new parallel sub-tasks to which it assigns workers for the parallel-execution portion of the request (if we impose on a workload a maximum degree of parallelism of one, SQL Server will still generate a parallel plan for execution but only allow the parent task to execute the parallel operations in the plan).

Normally, there is one SQLOS scheduler per CPU "core". For example, a 16-core processor has 16 physical CPU and so 16 schedulers. An 8-core processor with hyper-threading has 8 physical CPU and 8 logical CPU and, again, 16 schedulers. Only one session's worker thread can be running on the CPU at any given time. A thread can be in one of three states, as exposed by the `STATUS` column of the `sys.dm_exec_requests` DMV:

- **`RUNNING`** – on the CPU.
- **`SUSPENDED`** – whenever a thread requires a resource, such as a page that is not in memory, the thread yields the CPU and moves onto an unordered `waiter list`, with a thread status of `SUSPENDED`, until the required resource is available.
- **`RUNNABLE`** – if a thread is not waiting on any resource, but is not currently on the CPU, for example because the scheduler to which it is assigned currently has another session's thread running, it will be placed in a first-in-first-out (FIFO) queue called the runnable queue and the thread status will be RUNNABLE.

The SQLOS allows any given thread a small quantum of processing time (4 ms), before it must yield the CPU to another session's thread. If, before its allotted quantum is complete, the running thread needs to request a resource, it moves to the waiter list. Once the resource is available, the thread moves to the back of the runnable queue and then, when its time comes, back onto the CPU. It then gets another 4 ms quantum of processing time. If it doesn't have to wait for any resource in this time, but fails to complete its work, it yields the CPU anyway, moves to the back of the runnable queue, and awaits its next turn.

Resource waits and signal waits

Ideally, most of the time required to process a user request will be time spent by its thread on the CPU. This is all pure CPU time, or **service time**. Of course, a complex query requiring lots of data, or very complex aggregations and calculations, or both, will require longer service time than a small, simple query. SQL Server may decide to **parallelize** processing of a complex request across several worker threads, and so split the work over several CPUs, if available.

However, there are many reasons why processing of a task, parallel or otherwise, may not be able to proceed uninterrupted. A request that is on the CPU may be unable to proceed processing because it needs to wait for a particular resource to become available. For example, it may need to wait for a data page to be read from disk into the buffer cache, or for another request to release a lock on a data page to which it needs access. For a data modification statement, it may need to wait to write the changes, synchronously, to the database's transaction log, on disk. As soon as a running thread requires a resource, it is suspended (i.e. it moves to the **waiter list**). This is known as a **resource wait**. Every time a request's worker thread has to wait for a resource, the SQLOS tracks the time spent waiting (the **wait time**) as well as the cause of the wait, called the **wait type**, which is the resource on which the thread waited.

If a thread is ready to run (has been 'signaled') but is in the runnable queue, the SQLOS tracks the time spent waiting to get back on the CPU. This is referred to as a signal wait and is pure CPU wait. If query response times are slow and we find mainly **signal waits**, then we may need to focus our efforts on optimizing CPU usage.

The **total wait time** for a thread is the time elapsed between yielding the CPU and getting back on it, in other words the resource wait time plus the signal wait time.

SQLOS's wait statistics DMVs

The SQLOS exposes this wait statistics data primarily through two dedicated `sys.dm_os_*` DMVs.

sys.dm_os_waiting_tasks

The `sys.dm_os_waiting_tasks` DMV is a view onto the **waiter list**. In it, the SQLOS records the wait statistics, i.e. wait types and wait times, for any requests associated with a currently suspended thread, i.e. any thread that is in the waiter list, waiting for a resource to become available.

Therefore, we can use this DMV to investigate the primary **resource wait types** for a server currently experiencing performance issues (wait times and types for currently blocked requests also appear in `sys.dm_exec_requests`).
One common, but not the only, wait type that will show up in `sys.dm_os_waiting_tasks` is lock-related waits, where for example one long-running request blocks many other requests for a significant period.

sys.dm_os_wait_stats

The **sys.dm_os_wait_stats** DMV provides the historical perspective of the most significant waits associated with the overall workload on the server. To give just one example (we'll look at many more later), every time a thread, for any request, has to wait while SQL Server reads a data page into the buffer cache, then SQLOS adds the recorded wait time to the total cumulative time for a wait of type `PAGEIOLATCH_SH`.

In this DMV, the SQLOS records the wait type and the cumulative total wait time for each type of wait. The term **cumulative total wait time** needs a little elaboration:

- **Cumulative** – for waits of each type the DMV stores the cumulative wait times, across all threads and all sessions, since the last instance restart, or since the last time someone cleared out the wait statistics manually.
- **Total** – the recorded wait times include both the time spent by a thread in the waiter list (resource wait) and the time it spends in the runnable queue (signal wait).

The `sys.dm_os_wait_stats` DMV stores both the cumulative total wait time in milliseconds (`wait_time_ms`) as well as the cumulative signal wait, i.e. the time spent in the RUNNABLE queue (`signal_wait_time_ms`). Therefore, we can calculate the cumulative time threads spent in the `SUSPENDED` list, waiting for a particular resource, as follows:

```
Resource waits = Total waits – Signal waits
               =(wait_time_ms) - (signal_wait_time_ms).
```

We can use `sys.dm_os_wait_stats` to investigate the most prevalent resource wait types over a given period, and to investigate CPU waits, possibly indicating CPU pressure.

When using `sys.dm_os_wait_stats`, it's very hard to diagnose an ailing system just by examining the current state of the historical wait data, which may have accumulated over weeks or even months. The waits currently causing issues on the system could easily be 'hidden' among the accumulated waits. Therefore, it's important to track these statistics over time, establish the normal "baseline" for wait types for your workload and then look for anomalies. We'll explain how to capture waits statistics baselines later in this paper.

Using wait statistics for performance tuning

Analysis of wait statistics represents a very effective means to diagnose response times in our databases. In very simple terms, our requests either work, or they wait:

```
Response time = service time + wait time
```

If we find the response time comprised mainly time spent waiting for a particular resource (disk I/O, memory, network, and so on) then we know where to direct our tuning efforts. However, there are two very important points to remember when using wait statistics for performance tuning.

First, it's perfectly normal for threads to wait during task execution. In fact, in busy databases supporting a high degree of concurrent access, it's almost inevitable that a thread for any particular task will have to wait for a resource at some point in its execution cycle, however briefly. If not, it may simply exceed its 4 ms quantum and wait in the runnable queue. Likewise, there are certain wait types that do not indicate a performance problem and that we can simply ignore during our wait statistics analysis (more on this later). What we're interested in is significant, and recurring, types of waits.

Second, wait statistics are not really a tool to use in isolation in order to diagnose a performance issue. They are best used in conjunction with other diagnostic tools, as a way to direct our tuning efforts and avoid wasting time in 'blind alleys'. We need to corroborate what we find from analyzing wait statistics with, for example, Windows and SQL Server performance monitor (PerfMon) counters, which provide specific resource usage measurements (disk transfer rates, amount of CPU time consumed, and so on), which highlight the "queues" in our database system. By correlating wait statistics with resource measurements, you can quickly locate the most contested resources on your system, and so highlight potential bottlenecks.

> ***SQL Server 2005 waits and queues***
> *The use of "waits and queues" as the basis for a performance tuning methodology is explained in an excellent white paper by Tom Davidson, which is available here: http://download.microsoft.com/download/4/7/a/47a548b9-249e-484c-abd7-29f31282b04d/Performance_Tuning_Waits_Queues.doc.*

Investigating active-but-blocked requests using sys.dm_os_waiting_tasks

We can investigate the current, "real time" wait profile of a SQL Server instance, in terms of the requests and sessions associated with suspended threads, by querying the **`sys.dm_os_waiting_tasks`** DMV and joining it to various other useful views and functions. The `sys.dm_os_waiting_tasks` DMV reveals, amongst other columns, the:

- **`session_id`** of the session associated with the suspended thread.
- **`execution_context_id`** of the task associated with the suspended thread. If there is more than one `execution_context_id` associated with a single `session_id` then SQL Server has parallelized the task.
- **`wait_type`** – the current wait type for the suspended thread.
- **`wait_duration_ms`** – length of time, in ms, that the suspended thread has waited for the current wait type.
- **`blocking_session_id`** – reveals the `session_id` associated with the blocking thread, if a thread is suspended due to blocking, for example because it is waiting to acquire a lock.
- **`resource_description`** – for some types of resource, a description of the resource the suspended thread is waiting to acquire. For example, if it's a locking wait, this reveals the lock level (page, table, and so on) and the `ID` of the locked resource.

As noted previously, many wait types are fleeting and so querying the `waiting_tasks` DMV repeatedly, for a normally-functioning server, will likely reveal an ever-changing profile of wait types. This DMV is most useful when a system is currently experiencing blocking issues, with one or more sessions waiting for resources that other sessions hold.

In such cases, we will likely observe significant **locking waits** (waits of type `LCK_M_XX`), indicating that a session is unable to proceed with its work until it can acquire a lock on the resource on which another session's thread holds a conflicting lock. The most common causes of persistent blocking are poor code, which causes SQL Server to hold locks for far longer than necessary, lack of indexing, or an IO-intensive process that is blocking many other sessions for long periods. (Of course, if the server under investigation has a persistent history of blocking problems, then `LCK_*` waits will also show up prominently in `sys.dm_os_wait_stats`.)

Listing 1 queries the `sys.dm_os_waiting_tasks` DMV to show all waiting tasks **currently** active or blocked, revealing the wait type, duration, and resource. With a join to the `sys.dm_exec_requests` DMV, on the `blocking_session_id`, and `CROSS APPLY` to `sys.dm_exec_sql_text`, we also reveal the identity of the blocked and blocking sessions and the associated statements.

```
SELECT  blocking.session_id AS blocking_session_id ,
        blocked.session_id AS blocked_session_id ,
        waitstats.wait_type AS blocking_resource ,
        waitstats.wait_duration_ms ,
        waitstats.resource_description ,
        blocked_cache.text AS blocked_text ,
        blocking_cache.text AS blocking_text
FROM    sys.dm_exec_connections AS blocking
        INNER JOIN sys.dm_exec_requests blocked
                ON blocking.session_id = blocked.blocking_session_id
        CROSS APPLY sys.dm_exec_sql_text(blocked.sql_handle)
                                                    blocked_cache
        CROSS APPLY sys.dm_exec_sql_text(blocking.most_recent_sql_handle)
                                                    blocking_cache
        INNER JOIN sys.dm_os_waiting_tasks waitstats
                ON waitstats.session_id = blocked.session_id
```

Listing 1: Investigating blocking using the `sys.dm_os_waiting_tasks` DMV

Using the `blocking_session_id`, we can follow the blocking chain and find out what request is at the head of the chain and therefore the root cause of all the locking waits. We could easily extend Listing 1 to return the execution plan for the blocked session simply by joining (using the `APPLY` operator) to the `sys.dm_exec_query_plan` function and passing in the `plan_handle` from `sys.dm_exec_requests`.

In the sample output shown in Figure 1, we see that one session (`51`) is blocking another (`56`) from attaining a shared read lock (`LCK_M_S`) on a particular page (`14985`).

	blocking_session_id	blocked_session_id	blocking_resource	wait_duration_ms	resource_description	blocked_text
1	51	56	LCK_M_S	949303	pagelock fileid=1 pageid=14985 dbid=23 id=lock87...	SELECT FirstNa

Figure 1: A locking wait in the `sys.dm_os_waiting_tasks` output

We can see the SQL statements executed by both the blocked and blocking sessions. We can also use the `resource_description` to find out, for example, the identity of the page that is the source of blocking contention. Armed with the `pageid` of the contended page, we could use `DBCC PAGE` to find the owning table and index.

From there, we can investigate the index structures to try to find out what might be causing the blocking. For example, we can return further information about locking, latching, and blocking for specific indexes from `sys.dm_db_index_operational_stats`. We won't cover that here, but Chapter 5 of the book SQL Server Performance Tuning with DMVs explains how to return and analyze data from this view.

In this manner, we can pinpoint and resolve the root cause of the blocking. Commonly, we might find that the head of the chain is a substantial update request, or table scan, which causes an entire table to be locked and so all other requests on that table to be blocked for the duration. Other times, we might find a 'stalled' IO operation, pointing to potential disk IO issues.

It can be useful to corroborate the wait statistics with good baselines for locking related PerfMon counts such `Lock Waits/sec` and `Avg Wait Time (ms)`, and possibly `Lock requests/sec`, so we can gauge whether the level of locking activity, and lock waits times, is abnormal.

Ultimately, resolution of waits related to locking is an exercise in query tuning and index strategy. It is well beyond the scope of this whitepaper to cover those topics here, but Chapter 6 of the Troubleshooting SQL Server book offers some good starting points, as well as further troubleshooting and reporting techniques for blocking issues.

See Further Reading for full details of the books referenced in this section, and other useful resources.

Analyzing historical wait statistics using sys.dm_os_wait_stats

We can also perform historical wait statistics analysis, using the data provided in `sys.dm_os_wait_stats`. The wait times provided by `sys.dm_os_wait_stats` are running totals, accumulated across all threads and sessions since the server was last restarted or the statistics were manually reset using the `DBCC SQLPERF` command, shown in Listing 2.

```
DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);
```

Listing 2: Resetting the wait statistics

If a SQL Server instance has been running for quite a while and then is subjected to a significant change, such as adding an important new index, it's worth considering clearing the old wait stats in order to prevent the old cumulative wait stats masking the impact of the change on the wait times.

The `sys.dm_os_wait_stats` DMV gives us a list of all the different types of waits that threads have encountered, the number of times they have waited on a resource to be available, and the amount of time waited. The following columns are available in the view (times are all in millisecond units):

- **`wait_type`** – the type of wait, which generally indicates the resource on which the worked threads waited (typical resource waits include lock, latch disk I/O waits, and so on).
- **`wait_time_ms`** – total, cumulative amount of time that threads have waited on the associated wait type; this value includes the time in the **`signal_wait_time_ms`** column. The value increments from the moment a task stops execution to wait for a resource, to the point it resumes execution.
- **`signal_wait_time_ms`** – the total, cumulative amount of time threads took to start executing after being signaled; this is time spent on the runnable queue.
- **`waiting_tasks_count`** – the cumulative total number of waits that have occurred for the associated resource (`wait_type`).
- **`max_wait_time_ms`** – the maximum amount of time that a thread has been delayed, for a wait of this type.

There are many reasons why a certain task within SQL Server may need to wait, which means there are many possible values for the `wait_type` column. Some are quite usual, such as the need to wait for a lock to be released before a task can access the required resource (e.g. a data page), and are indicated by the "normal" lock modes such as shared, intent, and exclusive. Other common causes of waits include latches, backups, external operations such as extended stored procedure execution, replication, and resource semaphores (used for memory access synchronization). There are too many to cover them all in detail, though a little later, we'll examine many of the most common wait types and what they indicate. All of the wait types are at least listed, if not well documented, in Books Online. Alternatively, some third party monitoring tools, such as Red Gate SQL Monitor, maintain a useful list of wait types, and possible causes, as part of their tool documentation.

Identifying high signal waits (CPU pressure)

If the signal wait time is a significant portion of the total wait time then it means that tasks are waiting a relatively long time to resume execution after the resources that they were waiting for became available. This can indicate either that there are lots of CPU-intensive queries, which may need optimizing, or that the server needs more CPU. The query in Listing 3 will provide a measure of how much of the total wait time is signal wait time.

```
SELECT  SUM(signal_wait_time_ms) AS TotalSignalWaitTime ,
        ( SUM(CAST(signal_wait_time_ms AS NUMERIC(20, 2)))
          / SUM(CAST(wait_time_ms AS NUMERIC(20, 2))) * 100 )
                        AS PercentageSignalWaitsOfTotalTime
FROM    sys.dm_os_wait_stats
```

Listing 3: Identifying possible CPU pressure via signal wait time

The `sys.dm_os_wait_stats` DMV shows aggregated wait times and counts starting from when the wait statistics were cleared, or from when the server started. Therefore, a "point-in-time" view of the wait stats is generally not that useful. What is most useful is to compare the wait stats at a particular time with the stats at an earlier time and see how they changed. The other option is to clear the wait stats DMV (see Listing 2), and wait a set period, and then query and see what's accumulated in that time.

If signal waits analysis indicates CPU pressure, then the `sys.dm_os_schedulers` DMV can help verify whether a SQL Server instance is currently CPU-bound. This DMV returns one row for each of the SQL Server schedulers and it lists the total number of tasks that are assigned to each scheduler, as well as the number that are runnable. Other tasks on the scheduler that are in the `current_tasks_count` but not the `runnable_tasks_count` are ones that are either sleeping or waiting for a resource.

```
SELECT  scheduler_id ,
        current_tasks_count ,
        runnable_tasks_count
FROM    sys.dm_os_schedulers
WHERE   scheduler_id < 255
```

Listing 4: Investigating scheduler queues

The filter for schedulers below 255 removes from the result set the numerous hidden schedulers in SQL Server, which are used for backups, the Dedicated Administrator Connection (DAC), and so on, and are not of interest when investigating general CPU load.

There is no threshold value that represents the boundary between a "good" and "bad" number of runnable tasks, but the lower the better. A high number of runnable tasks, like a high signal wait time, indicate that there is not enough CPU for the current query load. If the scheduler queue is currently long, it's likely you'll also see the `SOS_SCHEDULER_YIELD` wait type in queries against `sys.dm_exec_requests` and `sys.dm_os_waiting_tasks`.

You'll also want to collect and analyze supporting data from the PerfMon counters that are of value for monitoring CPU usage, listed below with brief explanations (quoted from MSDN):

- **`Processor/ %Privileged Time`** – "percentage of time the processor spends on execution of Microsoft Windows kernel commands such as core operating system activity and device drivers."
- **`Processor/ %User Time`** – "percentage of time that the processor spends on executing user processes such as SQL Server. This includes I/O requests from SQL Server"
- **`Process (sqlservr.exe)/ %Processor Time`** – "the sum of processor time on each processor for all threads of the process".

If you confirm CPU pressure, probably the first step is to determine the most CPU-intensive queries in your workload, and see if there are ways to tune them. It is beyond the scope of this booklet to discuss this topic in detail but an excellent resource is the *Troubleshooting SQL Server, A Guide for the Accidental DBA* book. Chapter 3 in this book shows how to interrogate the `sys.dm_exec_query_stats` DMV to find the most costly queries in cache by total worker time (i.e. CPU time), joining to the `sys.dm_exec_sql_text` and `sys.dm_exec_query_plan` functions to retrieve the text and execution plans for these queries.

Identifying the primary resource waits

The primary use of the `sys.dm_os_wait_stats` DMV is to help us determine on which wait types and which resources SQL Server has spent the most time waiting, historically, while executing that server's workload. As noted earlier, it's important to focus this analysis on the most prevalent waits, and also filter out 'benign' waits that will occur frequently on any database system and are not, except in extreme edge cases, the cause of performance issues.

Listing 5 is reproduced with permission from the original author, Paul Randal. It provides both total wait times as well as resource and signal wait times, filters out many benign waits, and performs some math to present very clearly the most significant waits, historically, on the instance. Periodically, Paul amends and improves the query in his original post, for example to expand the benign waits filter. It's worth checking his post for the latest version.

This listing will help us locate the biggest bottlenecks, at the instance level, allowing us to focus our tuning efforts on a particular type of problem. For example, if the top wait types are all disk I/O related, then we would want to investigate this issue further using disk-related DMV queries and PerfMon counters.

```
WITH  [Waits]
        AS ( SELECT    [wait_type] ,
                       [wait_time_ms] / 1000.0 AS [WaitS] ,
                       ( [wait_time_ms] - [signal_wait_time_ms] ) / 1000.0
                                                    AS [ResourceS] ,
                       [signal_wait_time_ms] / 1000.0 AS [SignalS] ,
                       [waiting_tasks_count] AS [WaitCount] ,
                       100.0 * [wait_time_ms] / SUM( [wait_time_ms] ) OVER ( )
                                                     AS [Percentage] ,
                       ROW_NUMBER () OVER ( ORDER BY [wait_time_ms] DESC )
                                                      AS [RowNum]
             FROM    sys.dm_os_wait_stats
             WHERE   [wait_type] NOT IN ( N'BROKER_EVENTHANDLER',
                                          N'BROKER_RECEIVE_WAITFOR',
                                          N'BROKER_TASK_STOP',
                                          N'BROKER_TO_FLUSH',
                                          N'BROKER_TRANSMITTER',
                                          N'CHECKPOINT_QUEUE', N'CHKPT',
                                          N'CLR_AUTO_EVENT',
                                          N'CLR_MANUAL_EVENT',
                                          <…wait type filter truncated.
                                          See code download…>
                   )
               )
    SELECT [W1].[wait_type] AS [WaitType] ,
           CAST ([W1].[WaitS] AS DECIMAL(16, 2)) AS [Wait_S] ,
           CAST ([W1].[ResourceS] AS DECIMAL(16, 2)) AS [Resource_S] ,
           CAST ([W1].[SignalS] AS DECIMAL(16, 2)) AS [Signal_S] ,
           [W1].[WaitCount] AS [WaitCount] ,
           CAST ([W1].[Percentage] AS DECIMAL(5, 2)) AS [Percentage] ,
           CAST (( [W1].[WaitS] / [W1].[WaitCount] ) AS DECIMAL(16, 4))
                                                     AS [AvgWait_S] ,
           CAST (( [W1].[ResourceS] / [W1].[WaitCount] ) AS DECIMAL(16, 4))
                                                     AS [AvgRes_S] ,
           CAST (( [W1].[SignalS] / [W1].[WaitCount] ) AS DECIMAL(16, 4))
                                                     AS [AvgSig_S]
     FROM [Waits] AS [W1]
          INNER JOIN [Waits] AS [W2] ON [W2].[RowNum] <= [W1].[RowNum]
     GROUP BY [W1].[RowNum] ,
              [W1].[wait_type] ,
              [W1].[WaitS] ,
              [W1].[ResourceS] ,
              [W1].[SignalS] ,
              [W1].[WaitCount] ,
              [W1].[Percentage]
     HAVING  SUM([W2].[Percentage]) - [W1].[Percentage] < 95; -- percentage
                                                              threshold
GO
```

Listing 5: Report on top resource waits

Common wait types

In general, when examining wait statistics, either historically or for currently active requests, it's wise to focus on the top waits, according to wait time, and look out for high wait times associated with the following specific wait types (some of which we'll cover in more detail in subsequent sections).

- **`CXPACKET`**
  Often indicates nothing more than that certain queries are executing with parallelism; `CXPACKET` waits in the server are not an immediate sign of problems, although they may be the symptom of another problem, associated with one of the other high value wait types on the instance.

- **`SOS_SCHEDULER_YIELD`**
  The tasks executing in the system are yielding the scheduler, having exceeded their quantum, and are having to wait in the runnable queue for other tasks to execute. As discussed earlier, this wait type is most often associated with high signal wait rather than waits on a specific resource. If you observe this wait type persistently, investigate for other evidence that may confirm the server is under CPU pressure.

- **`THREADPOOL`**
  A task had to wait to have a worker thread bound to it, in order to execute. This could be a sign of worker thread starvation, requiring an increase in the number of CPUs in the server, to handle a highly concurrent workload, or it can be a sign of blocking, resulting in a large number of parallel tasks consuming the worker threads for long periods.

- **`LCK_*`**
  These wait types signify that blocking is occurring in the system and that sessions have had to wait to acquire a lock, of a specific type, which was being held by another database session.

- **`PAGEIOLATCH_*, IO_COMPLETION, WRITELOG`**
  These waits are commonly associated with disk I/O bottlenecks, though the root cause of the problem may be, and commonly is, a poorly performing query that is consuming excessive amounts of memory in the server, or simply insufficient memory for the buffer pool.

- **PAGELATCH_***
  Non-IO waits for latches on data pages in the buffer pool. A lot of times PAGELATCH_* waits are associated with allocation contention issues. One of the most well-known allocations issues associated with PAGELATCH_* waits occurs in tempdb when the a large number of objects are being created and destroyed in tempdb and the system experiences contention on the Shared Global Allocation Map (SGAM), Global Allocation Map (GAM), and Page Free Space (PFS) pages in the tempdb database.

- **LATCH_***
  These waits are associated with lightweight short-term synchronization objects that are used to protect access to internal caches, but not the buffer cache. These waits can indicate a range of problems, depending on the latch type. Determining the specific latch class that has the most accumulated wait time associated with it can be found by querying the sys.dm_os_latch_stats DMV.

- **ASYNC_NETWORK_IO**
  This wait is often incorrectly attributed to a network bottleneck. In fact, the most common cause of this wait is a client application that is performing row-by-row processing of the data being streamed from SQL Server as a result set (client accepts one row, processes, accepts next row, and so on). Correcting this wait type generally requires changing the client side code so that it reads the result set as fast as possible, and then performs processing.

- **OLEDB**
  Occurs when a thread makes a call to structure that uses the OLEDB provider and is waiting while it returns the data. A common cause of such waits is linked server queries. However, for example, many DMVs use OLEDB internally, so frequent calls to these DMVs can cause these waits. DBCC checks also use OLEDB so it's common to see this wait type when such checks run.

These basic explanations of each of the major wait types won't make you an expert on wait type analysis, but the appearance of any of these wait types high up in the output of Listings 1 or 5 will certainly help direct your subsequent investigations.

After making a code or configuration change on the server, in response to a problem highlighted in the wait statistics, it's important to validate that the problem has indeed been fixed. In order to do this, we'd suggest resetting the wait statistics (Listing 2). One of the caveats associated with clearing the wait statistics on the server is that it will take a period of time for the wait statistics to accumulate to the point that you know whether or not a specific problem has been addressed.

The coming sections review some of the most common wait types associated with the server resources (Disk IO, CPU, and memory), along with common causes and possible routes towards resolution.

Disk IO-related waits

Three of the most prevalent disk IO-related waits are as follows:

- **`PAGEIOLATCH_XX`** – waits associated with delays in a thread being able to bring a data file page into cache from the physical data file
- **`WRITELOG`** – waits related to issues with writing to the log file
- **`ASYNC_IO_COMPLETION`** – waits associated with writing (asynchronously) to the data files

PAGEIOLATCH_XX

`PAGEIOLATCH_XX` waits appear when sessions have to wait to obtain a latch on a page that is not currently in memory. This happens when lots of sessions, or maybe one session in particular, request many data pages that are not available in the buffer pool, and so the engine needs to perform physical I/O to retrieve them. SQL Server must allocate a buffer page for each one and place a latch on that page while it retrieves it from disk.

PAGEIOLATCH_SH and PAGEIOLATCH_EX

If a thread needs to read a page, it needs a shared-mode latch on the buffer page, and waits for such latches for pages not currently in memory appear as ***`PAGEIOLATCH_SH`*** *wait types. If a thread needs to modify a page, we see* ***`PAGEIOLATCH_EX`*** *wait types.*

`PAGEIOLATCH_XX` waits are one of the hardest to diagnose correctly or, to put it another way, one of the easiest to misdiagnose, since it sounds like a symptom of a malfunctioning or underpowered **disk subsystem**. This could be the cause, but the cause is just as likely to be **insufficient memory** to accommodate the workload, and even more likely to be **poorly tuned queries** driving far more IO than is necessary, which we can often fix with query tuning and indexes.

`PAGEIOLATCH_XX` waits do indicate that disk I/O is the bottleneck, since it ultimately means that the disk subsystem can't return pages quickly enough to satisfy all of the page requests, and so sessions are waiting to acquire these latches and performance is suffering. However, the appearance of this wait type does not necessarily indicate that a slow disk subsystem is the *cause* of the bottleneck; it may simply be the victim of excessive IO caused by a problem elsewhere in the system.

Gathering supporting evidence: virtual file statistics and PerfMon counters

Before jumping to any conclusions on the root cause, we should evaluate `PAGEIOLATCH_XX` waits in conjunction with **virtual file statistics**, using the `sys.dm_io_virtual_file_stats` dynamic management function, as well as other prevalent wait types and data from performance counters relating to disk latency and memory.

For example, let's say we identify very high `PAGEIOLATCH_XX` waits, from either of the SQLOS wait statistics DMVs. Our next step might be to examine the overall read and write patterns and the distribution of IO load across files and databases on the SQL Server instance to see if we can find any obvious "hotspots". Alternatively, if a server is currently experiencing high `PAGEIOLATCH_XX` waits, we can use `sys.dm_os_waiting_tasks` to identify the page and objects that are the target of threads experiencing these waits, and then analyze the latencies on the relevant files for those objects.

A query such as that shown in Listing 6 will help us determine if the problem is specific to a single database, file, or disk, or is instance wide.

```
SELECT  DB_NAME(vfs.database_id) AS database_name ,
        vfs.database_id ,
        vfs.file_id ,
        io_stall_read_ms / NULLIF(num_of_reads, 0) AS avg_read_latency ,
        io_stall_write_ms / NULLIF(num_of_writes, 0)
                                            AS avg_write_latency ,
        io_stall_write_ms / NULLIF(num_of_writes + num_of_writes, 0)
                                            AS avg_total_latency ,
        num_of_bytes_read / NULLIF(num_of_reads, 0)
                                            AS avg_bytes_per_read ,
        num_of_bytes_written / NULLIF(num_of_writes, 0)
                                            AS avg_bytes_per_write ,
        vfs.io_stall ,
        vfs.num_of_reads ,
        vfs.num_of_bytes_read ,
        vfs.io_stall_read_ms ,
        vfs.num_of_writes ,
        vfs.num_of_bytes_written ,
        vfs.io_stall_write_ms ,
        size_on_disk_bytes / 1024 / 1024. AS size_on_disk_mbytes ,
        physical_name
FROM    sys.dm_io_virtual_file_stats(NULL, NULL) AS vfs
        JOIN sys.master_files AS mf ON vfs.database_id = mf.database_id
                                    AND vfs.file_id = mf.file_id
ORDER BY avg_total_latency DESC
```

Listing 6: Virtual file statistics

This will help us identify any files or databases associated with high volumes of reads or writes, and IO latencies. In particular, we're seeking evidence of high latency associated with read and write operations to a particular file or database, via the `IO_STALL_*` columns. To corroborate this data further, we might also check the drive-level disk IO values using the PerfMon counters, `Physical Disk\Avg. Disk sec/Read` and `Physical Disk\Avg. Disk sec/Write`. Latency values over about 10ms are generally a concern, though of course, "it depends".

With this information to hand, we'll want to investigate further the activity on files or databases identified as IO hotspots and determine the root cause of the waits.

Possible causes of PAGEIOLATCH waits: sub-optimal queries

Often, the problem is inefficient queries, query plans, and indexes. If a query is performing large scans instead of targeted seeks, it is potentially reading many more pages than necessary and will likely cause excessive physical as well as logical IO, as SQL Server needs to read large numbers of pages in from disk. SQL Server will often parallelize these big scans, so it's quite common to see `PAGEIOLATCH_XX` waits in conjunction with `CXPACKET` waits (see later). It will also lead to overuse of the buffer cache.

Having established the databases and files that are physical IO hotspots, and having identified requests and objects currently associated with the `PAGEIOLATCH_XX` waits, using `sys.dm_os_waiting_tasks` (Listing 1), we can investigate the execution statistics for queries against the relevant database. These are maintained in `sys.dm_exec_query_stats` for the execution plans that are in the plan cache. We need to identify the queries that have the highest accumulated physical reads, and then review their associated execution plans, looking for any performance tuning opportunities, either by adding missing indexes to the database, or making changes to the SQL code, in order to optimize the way the database engine accesses the data.

A particular issue to look for in the execution plans is implicit conversions, which will prevent the optimizer using indexes to resolve the query, and result in scans, rather than seeks. We also need to look out for issues such as out-of-date statistics, causing the optimizer to choose suboptimal plans, again, often resulting in inefficient scans, and SQL Server reading more pages than necessary.

Possible causes of PAGEIOLATCH waits: memory pressure

Sometimes, the root cause of `PAGEIOLATCH_XX` waits is a 'memory problem' in that the buffer pool is experiencing memory pressure, because it is sized inadequately, or it is being "overused" (see previous section).

In order to verify this, we can examine the PerfMon memory counters (see later). If we see that the `Page Life Expectancy` is low or consistently fluctuating, and the system is experiencing non-zero values for `Free List Stalls/sec`, and high `Lazy Writes/sec`, then we can be fairly certain that the buffer pool is experiencing memory pressure, meaning that it cannot retain pages in memory for any reasonable length of time. This can lead to cache "churn" and a high level of disk IO activity.

If we observe `PAGEIOLATCH_XX` waits in conjunction with memory pressure on the buffer pool, we need to investigate the cause of the pressure. It could be due to external pressure (from the Windows OS) or internal pressure, meaning that some other SQL Server resource is eating memory that would otherwise be allocated to the buffer pool. A common cause of the latter is a bloated plan cache, due to ad-hoc non-parameterized queries and the production of many single-use plans.

Of course, sometimes, we might find that the code is optimized as far as possible, but a database that used to be 1GB size and fit comfortably in memory is now 4GB in size and we've not expanded the server's memory capacity appropriately. If a common reporting query needs to read most or all of that 4 GB of data from disk, for aggregation, it will cause high physical I/O and account for the high `PAGEIOLATCH_XX` waits. At this point, the server needs more memory.

Possible causes of PAGEIOLATCH waits: disk subsystem issues

Finally, we occasionally find that the root cause really is slow storage and we need to consider hardware upgrade, or workload redistribution.

For example, let's say that our virtual file statistics highlight that the IO 'hotspot' is a database used for archiving data to slower storage, for year-on-year reporting. In such cases, it is predictable and expected that we will see `PAGEIOLATCH_XX` waits in the database during the archival period.

If the archiving process is as tuned as possible, but its effect on other business process has become intolerable, then we may indeed have a poorly configured or underpowered disk subsystem. If so, we could consider workload redistribution, moving the 'hot' files to faster disks.

WRITELOG

SQL Server's write-ahead logging mechanism guarantees the durability of transactions by writing transaction details, synchronously, to the transaction log when each transaction commits. Thus, when servicing a data modification request, SQL Server writes a description of the changes to log records, in the log buffer, and modifies the relevant data pages, in memory. When all necessary changes are complete, and SQL Server is ready to commit the transaction, it first hardens to disk all of the log records in the buffer, up to the point of the `COMMIT`. Note that this is not a selective flushing to disk of just the log records relating to the current transaction, but a flushing of all log records up to that point, regardless of whether there are other log records associated with as yet uncommitted transactions. Only later, at regular `CHECKPOINT` operations does SQL Server, in general, write the changed data pages to disk.

On high-volume, write-intensive systems it's possible that the IO subsystem will fail to cope with the volume of log writes, resulting in `WRITELOG` waits, which essentially means that a thread is waiting for a transaction log block buffer to flush to disk. This, of course, is a synchronous operation; if a request requires a log buffer flush, either by issuing a commit or because the current log block is full, SQL Server must complete and acknowledge this operation before the request can proceed.

`WRITELOG` waits often arise when the log files for multiple, highly transactional databases are on the same physical disks. In such cases, the waits are caused by the disk head having to be repositioned across the disk platters to perform the writes, and can be worse when multiple transaction log files are physically interleaved, or physically fragmented on disk.

Again, if you see `WRITELOG` waits in `sys.dm_os_wait_stats`, you'll want to correlate the observed wait times with the IO latency values for the log file in `sys.dm_io_virtual_file_stats`.

Internal log flush limits

In his article, Wait statistics, or please tell me where it hurts, Paul Randal explains how ***`WRITELOG`*** *waits may arise due to exceeding "internal log flush limits" (indicated by WRITELOG times that are longer than IO latency values for writing to the log file). He suggests that may mean "splitting your workload over multiple databases or even make your transactions a little longer to reduce log flushes".*

Ultimately, `WRITELOG` could indicate a need to review your transaction log architecture, but it could equally be that that instance is simply generating a lot of unnecessary log writes due, for example, to log records generated when maintaining unused non-clustered indexes, or excessive index rebuilds, or by constant page splits as indexes created with very high fill factor start to fragment. See Paul Randal's post, Trimming the Transaction Log Fat, for further details of how you can reduce unnecessary log writes.

IO_COMPLETION and ASYNC_IO_COMPLETION

These waits occur when there is a delay in either a synchronous or asynchronous IO operation.

`IO_COMPLETION` waits often show up during operations such as loading a DLL from disk, writing a `tempdb` sort file, during certain `DBCC CHECKDB` operations, sparse file flushes, and more. The key is that it is IO but not typically database file IO for reading or writing pages or log records (`WRITELOG`).

The `ASYNC_IO_COMPLETION` wait type often shows up during sizeable `BACKUP` operations, for example. The fact that SQL Server writes the data asynchronously often means that these waits are often less of a concern in terms of performance that `WRITELOG` waits.

CPU-related waits

Most of time we investigate CPU pressure via signal waits and PerfMon counters as discussed previously. However, two interesting wait types to look out for, in regard to CPU pressure, are `CXPACKET` and `SOS_SCHEDULER_YIELD`.

Others, which we won't discuss in detail include (explanation quoted from BOL):

- **`THREADPOOL`** – "Occurs when a task is waiting for a worker to run on. This can indicate that the maximum worker setting is too low, or that batch executions are taking unusually long, thus reducing the number of workers available to satisfy other batches". It usually occurs during highly-parallelized workloads.
- **`CMEMTHREAD`** – "Occurs when a task is waiting on a thread-safe memory object" (i.e. one that must be accessed one thread at a time, typically a cache).

CXPACKET

`CXPACKET` waits occur during synchronization of the query processor exchange iterator between workers, for a query running in parallel across multiple processors. This wait type is set whenever a thread for a parallel process has to wait in the exchange iterator for another thread to complete, before it can continue processing. Often, the appearance of `CXPACKET` waits indicates nothing more than the fact that SQL Server is parallelizing queries, since the coordinator thread in a parallel query will always accumulate these waits, as it waits for each thread running a sub-task to complete its work.

Parallel workloads, and the associated parallelism operations that the server must support, can generate very heavy demands for memory allocation and disk IO. If the server hosts a data warehouse or reporting type of database that receives a low volume of queries but processes large amounts of data, parallelism can substantially reduce the time it takes to execute those queries. In contrast however, if the server hosts an OLTP database that has a very high volume of small queries and transactions, then parallelism can degrade both throughput and performance.

A common diagnosis on seeing very high `CXPACKET` waits is that the parallel workload is overwhelming the disk IO subsystem and a common response, in lieu of upgrading the disk subsystem, is to reduce the `max degree of parallelism` to one half or one fourth the number of logical processors, or processor cores, on the server, or even to completely disable parallelism by setting it to 1. While this is *sometimes* necessary, it is not the best response in all, or even most, situations.

If you're seeing lots of parallelization, and attendant `CXPACKET` waits in an OLTP system, the first question to ask is why SQL Server is parallelizing so many of your "short, fast" transactions. It is worth investigating those queries that are getting parallel plans and looking for tuning possibilities. It may also be that the `cost threshold for parallelism` setting is a little low for your system and raising it will remove some of the inappropriate parallelization.

For workloads where parallelization is appropriate, often `CXPACKET` waits are simply a symptom of a different underlying problem rather than an indication that parallelization is the actual problem. The trick is to isolate and troubleshoot the main underlying non-`CXPACKET` wait type that is causing the `CXPACKET` waits. Of course, the fact that the underlying wait causes many parallel workers to wait means that the volume of `CXPACKET` waits will often greatly exceed the underlying root wait type.

The underlying problem could be, for example, that there is an uneven distribution of work among the threads, or that one thread gets persistently blocked so that `CXPACKET` waits accumulate for other worker threads in the parallel process. If we can resolve the underlying wait, from the historical wait statistics, then we can remove the problem without having to limit the ability of SQL Server to parallelize query execution. If the system is currently experiencing high `CXPACKET` waits, as evidenced by `sys.dm_os_waiting_tasks`, look closely at associated waits for the active sessions for signs of an `LCK*` waits or `PAGEIOLATCH_*` waits and so on.

For example, let's say the underlying wait type, is `PAGEIOLATCH_XX` meaning that the parallel operation is waiting on a read from the disk IO subsystem (possibly you will also see accompanying `IO_COMPLETION, ASYNC_IO_COMPLETION` waits). If we diagnose the `PAGEIOLATCH_XX` waits, in the manner described previously, we resolve the problem. It could be that the disk subsystem simply can't cope with the demands of the parallel workload, but it could equally be that missing indexes or a bad query plan are causing parallel table scans and consequently much more disk IO than is necessary.

In such cases, reducing the `max degree of parallelism` *won't* resolve the root problem but it may buy you some time to fix the real problem, since it will reduce the number of workers being used in the system, reduce the additional load the parallelism operations place on the disk IO subsystem, and reduce the accumulated wait time for the `CXPACKET` wait type. In the meantime, you can perform some tuning and indexing work, or scale up the IO performance of the server, as appropriate. If the workload is as tuned as it can be, and scaling up the disk subsystem is not possible, then reducing the level of parallelism to a degree that still allows parallel processing to occur without bottlenecking in the disk IO subsystem may be the best option to improve overall system performance.

It is possible that `CXPACKET` waits in conjunction with other wait types, for example `LATCH_*` and `SOS_SCHEDULER_YIELD`, does show that parallelism is the actual problem, and further investigation of the latch stats on the system will validate if this is actually the case. The `sys.dm_os_latch_stats` DMV contains information about the specific latch waits that have occurred in the instance, and if one of the top latch waits is `ACCESS_METHODS_DATASET_PARENT`, in conjunction with `CXPACKET, LATCH_*`, and `SOS_SCHEDULER_YIELD` wait types as the top waits, the level of parallelism on the system is the cause of bottlenecking during query execution, and reducing the '`max degree of parallelism' sp_configure` option may be required to resolve the problems.

SOS_SCHEDULER_YIELD

The SQL scheduler is a co-operative, non-preemptive scheduler. It relies on executing queries voluntarily relinquishing the CPU after a specific amount of running time (4 ms, as discussed earlier). When a task voluntarily relinquishes the CPU, and begins waiting to resume execution, the wait type assigned to the task is `SOS_SCHEDULER_YIELD`. The relinquishing task goes back onto the runnable queue and another task gets its allocated time on the CPU.

> ***Preemptive waits***
> *The only time a thread runs in preemptive mode, meaning Windows controls when it removes the thread from the CPU, not SQL Server, is when the thread makes a call to the Windows OS. It is possible for a thread to encounter preemptive waits (`PREEMPTIVE_OS_*`, where the `*` is a Windows API being called). The status of such a thread will be `RUNNING`, since in this mode SQL Server no longer controls its execution and so just assumes it is running.*

Since `SOS_SCHEDULER_YIELD` is not generally a resource wait, we won't see it in `sys.dm_os_waiting_tasks` (sometimes, it can be caused by spinlock contention, but we won't discuss that further here. See Paul Randal's blog post, http://www.sqlskills.com/blogs/paul/sos_scheduler_yield-waits-and-the-lock_hash-spinlock/ for more information).

If overall wait times are low, `SOS_SCHEDULER_YIELD` waits are benign. However, if queries show high wait times in `sys.dm_os_wait_stats` for the `SOS_SCHEDULER_YIELD` wait type, it's an indication of extremely CPU-intensive queries. If there are high wait times for this wait type overall on the server it can indicate either that there are lots of CPU-intensive queries, which may need optimizing, or that the server needs more CPU. Again, look out in general for any large, expensive scans. Chapter 3, *High CPU Utilization*, of the previously-referenced *Troubleshooting SQL Server* book investigates many possible causes of high CPU usage, including missing indexes, non-sargable predicates, parameter sniffing, implicit conversions and more (many of which also cause excessive IO).

If we see high `SOS_SCHEDULER_YIELD` waits but *no* associated `PAGEIOLATCH_*` waits, this is more an indication that threads are persistently using up the allotted quantum of CPU time without completing their task, and could indicate a need for higher CPU power.

Scheduler activity can be investigated further using the `sys.dm_os_schedulers` DMV, as discussed previously.

Memory waits

One of the memory-related waits that we see often is the `PAGELATCH_XX` wait, which occurs when a thread is forced to wait for access to a data page that is in the buffer pool.

The best known cause of this wait type is contention for allocation pages (PFS, GAM, SGAM) during highly concurrent access of `tempdb`.

> ***Tip: Configuring a single file per processor, or not!***
> *SQL Server MVP Paul Randal provides guidance on the PFS, GAM, SGAM problem with* `tempdb` *and the appropriate number of files to configure based on the number of processors in his blog posts:*
> *Search Engine Q&A #12: Should you create multiple files for a user DB on a multi-core box? (http://www.sqlskills.com/blogs/paul/post/Search-Engine-QA-12-Should-you-create-multiple-files-for-a-user-DB-on-a-multi-core-box.aspx) and A SQL Server DBA myth a day: (12/30) tempdb should always have one data file per processor core (http://sqlskills.com/BLOGS/PAUL/post/A-SQL-Server-DBA-myth-a-day-(1230)-tempdb-should-always-have-one-data-file-per-processor-core.aspx).*

However, these waits can also occur when there is a high level of concurrent `INSERT` activity on a table that uses a clustered index on an ever-increasing integer key (such as an `IDENTITY`), since it leads to contention for access to a single index page.

It can also occur in cases where inserts commonly lead to page splits, since to perform the split, SQL Server must acquire shared latches at all levels of the index, and then exclusive latches on the specific pages that need to be split.

Once again, we can use `sys.dm_os_waiting_tasks` to identify the target objects and the queries associated with threads experiencing this wait, and then investigate further for these types of issues.

The whitepaper Diagnosing and Resolving Latch Contention on SQL Server is a useful resource for further reading.

Wait Statistics baselines

It is extremely important to understand that all SQL Server instances will have waits regardless of how well we tune and optimize them. In the `sys.dm_os_wait_stats` DMV, the data can accumulate over a long period. If we then interrogate the DMV for, say, the "top 5 waits on a SQL Server instance", it will still be very difficult to know if any of these waits represent a potential problem, or are just "normal" for that instance. Our goal is to understand the usual waits for an instance.

Once we know what's normal, then we can focus our tuning efforts, and we have a reference in the event that performance suddenly degrades. We can see trends over time and look out for worrying change in the pattern of waits. We can also map sudden changes to, for example, deployment of new code, or change in hardware configuration and so on.

In order to establish what is normal, then track changes over times, we need to establish baselines for our historical wait statistics data. Here we offer a manual means to gather and maintain baseline data for historical wait statistics. However, tools exist that will gather and manage the same sort of baseline data for you, with less work by the DBA. In the Microsoft stack, Management Data Warehouse gathers this data, and a lot more, and some third party SQL Server monitoring tools, such as SQL Monitor, will gather baseline data, as well as help analyze it for trends and anomalies.

Clearing wait stats

By default, SQL Server clears the cumulative wait statistics for an instance from the `sys.dm_os_wait_stats` DMV upon instance restart. In addition, a DBA can clear the statistics manually using `DBCC SQLPERF` (see Listing 2).

It is not required to clear out wait statistics on a regular basis. However, in order to analyze this data in a meaningful way, both to understand what "normal" behavior is and also quickly spot abnormalities when comparing data from a previous period to current data, it is important that DBAs adopt a clear policy on the timing and frequency of clearing wait statistics.

If one DBA is clearing wait statistics at 6 AM daily, and another DBA is capturing the information at 7 AM, the data only represents the waits accumulated during an hour's workload, which may not represent the normal workload.

Ideally, we need to collect wait statistics for a period that represents normal activity, without clearing them. At the same time, however, the DBAs will want to understand how significant changes, such as adding a new index, or altering a configuration setting, affect the pattern of waits for the instance. Clearing the wait statistics immediately after making the change can help us understand the impact of the modification.

There are other reasons that may compel a DBA to clear waits statistics. For example, some companies use a single third-party utility for all backups, regardless of the application, resulting in an extended time to complete SQL Server database backups. While there are alternate options, which can perform very fast SQL Server backups (e.g. native SQL backups, dedicated third-party SQL backup applications), the DBA is unable to use them. In such cases, the DBA knows backup performance to be poor, but is unable to make improvements and instead may opt to clear out wait statistics after each backup job completes, to prevent any waits from the actual backup from influencing the interpretation of the wait statistics as a whole. Alternatively, the DBA can filter `BACKUP*` waits from the output.

The queries in Listing 7 will reveal when wait statistics were last cleared by an instance restart, as well as if, and when, someone last cleared them manually. Simply compare the two values to see if wait statistics have been manually cleared since the last restart.

```
SELECT  [wait_type] ,
        [wait_time_ms] ,
        DATEADD(SS, -[wait_time_ms] / 1000, GETDATE())
                                            AS "Date/TimeCleared" ,
        CASE WHEN [wait_time_ms] < 1000
             THEN CAST([wait_time_ms] AS VARCHAR(15)) + 'ms'
             WHEN [wait_time_ms] BETWEEN 1000 AND 60000
             THEN CAST(( [wait_time_ms] / 1000 )
                                            AS VARCHAR(15)) + ' seconds'
             WHEN [wait_time_ms] BETWEEN 60001 AND 3600000
             THEN CAST(( [wait_time_ms] / 60000 )
                                            AS VARCHAR(15)) + ' minutes'
             WHEN [wait_time_ms] BETWEEN 3600001 AND 86400000
             THEN CAST(( [wait_time_ms] / 3600000 )
                                            AS VARCHAR(15)) + ' hours'
             WHEN [wait_time_ms] > 86400000
             THEN CAST(( [wait_time_ms] / 86400000 )
                                            AS VARCHAR(15)) + ' days'
        END AS "TimeSinceCleared"
FROM    [sys].[dm_os_wait_stats]
WHERE   [wait_type] = 'SQLTRACE_INCREMENTAL_FLUSH_SLEEP';

/*
   check SQL Server start time - 2008 and higher
*/
SELECT  [sqlserver_start_time]
FROM    [sys].[dm_os_sys_info];


/*
   check SQL Server start time - 2005 and higher
*/
SELECT  [create_date]
FROM    [sys].[databases]
WHERE   [database_id] = 2
```

Listing 7: When were waits stats last cleared, either manually or by a restart?

Ultimately, it is at the discretion of the DBAs to decide when to clear out wait statistics, if at all. If in doubt, collect your wait statistics (see the next section) for a period that will capture a representative workload (for example one month). The next month, collect the stats on a more regular schedule, such as every Sunday, after scheduled code changes, and then immediately clear out the `sys.dm_os_wait_stats` DMV. Compare each of the four one-week data sets to the one-month set: do different wait patterns exist (for example, perhaps the last week of the month, when various business reports run, has different waits), or are they consistent across all five sets? If you see differences then you may want to consider clearing out the stats on a regular (e.g. weekly) basis.

Reviewing the collected wait stats is discussed in more detail in the *Reviewing Wait Statistics* section.

Creating the database

Let's start by creating the database to store this data. Please note that in Listing 8 you may need to change location, file size, and file growth settings, as appropriate for your environment, and that you will need to back up the database on a regular basis.

```
USE [master];
GO

CREATE DATABASE [BaselineData] ON PRIMARY
( NAME = N'BaselineData',
  FILENAME = N'M:\UserDBs\BaselineData.mdf',
  SIZE = 512MB,
  FILEGROWTH = 512MB
) LOG ON
( NAME = N'BaselineData_log',
  FILENAME = N'M:\UserDBs\BaselineData_log.ldf',
  SIZE = 128MB,
  FILEGROWTH = 512MB
);

ALTER DATABASE [BaselineData] SET RECOVERY SIMPLE;
```

Listing 8: Database to store baseline performance data

Collecting the data

In order to collect wait statistics, on a regular schedule, the first step is to create a table to hold the information, as shown in Listing 9 (as described previously, this script assumes the `BaselineData` database exists).

```
USE [BaselineData];
GO

IF NOT EXISTS ( SELECT  *
                FROM    [sys].[tables]
                WHERE   [name] = N'WaitStats'
                        AND [type] = N'U' )
    CREATE TABLE [dbo].[WaitStats]
        (
          [RowNum] [BIGINT] IDENTITY(1, 1) ,
          [CaptureDate] [DATETIME] ,
          [WaitType] [NVARCHAR](120) ,
          [Wait_S] [DECIMAL](14, 2) ,
          [Resource_S] [DECIMAL](14, 2) ,
          [Signal_S] [DECIMAL](14, 2) ,
          [WaitCount] [BIGINT] ,
          [Percentage] [DECIMAL](4, 2) ,
          [AvgWait_S] [DECIMAL](14, 2) ,
          [AvgRes_S] [DECIMAL](14, 2) ,
          [AvgSig_S] [DECIMAL](14, 2)
        );
GO

CREATE CLUSTERED INDEX CI_WaitStats
  ON [dbo].[WaitStats] ([RowNum], [CaptureDate]);
```

Listing 9: Creating the dbo.WaitStats table

The second step is simply to schedule a query to run on a regular basis, which captures the wait information to this table. Listing 10 derives from the resource waits query in Listing 5 (in turn adapted from Paul Randal's previously-referenced wait statistics post). This query uses a CTE to capture the raw wait statistics data, and then manipulates the output to include averages, for example, average wait (`AvgWait_S`), and average signal wait (`AvgSig_S`). It includes an additional, optional `INSERT` as it helps to separate each set of data collected when reviewing the output (see the code download package for the full listing).

```
USE [BaselineData];
GO

INSERT  INTO dbo.WaitStats
        ( [WaitType]
        )
VALUES  ( 'Wait Statistics for ' + CAST(GETDATE() AS NVARCHAR(19))
        );

INSERT  INTO dbo.WaitStats
        ( [CaptureDate] ,
          [WaitType] ,
          [Wait_S] ,
          [Resource_S] ,
          [Signal_S] ,
          [WaitCount] ,
          [Percentage] ,
          [AvgWait_S] ,
          [AvgRes_S] ,
          [AvgSig_S]
        )
         EXEC
             ( '
       WITH [Waits] AS
          (SELECT
             [wait_type],
             [wait_time_ms] / 1000.0 AS [WaitS],
             ([wait_time_ms] - [signal_wait_time_ms]) / 1000.0
                                       AS [ResourceS],
             [signal_wait_time_ms] / 1000.0 AS [SignalS],
             [waiting_tasks_count] AS [WaitCount],
             100.0 * [wait_time_ms] / SUM ([wait_time_ms]) OVER()
                                       AS [Percentage],
             ROW_NUMBER() OVER(ORDER BY [wait_time_ms] DESC) AS [RowNum]
          FROM sys.dm_os_wait_stats
          WHERE [wait_type] NOT IN (
           N''BROKER_EVENTHANDLER'',    N''BROKER_RECEIVE_WAITFOR'',
           N''BROKER_TASK_STOP'',       N''BROKER_TO_FLUSH'',
           N''BROKER_TRANSMITTER'',     N''CHECKPOINT_QUEUE'',
           N''CHKPT'',                  N''CLR_AUTO_EVENT'',
           N''CLR_MANUAL_EVENT'',
           <…rest of filter truncated…>
)
        )
```

```
      SELECT
         GETDATE(),
         [W1].[wait_type] AS [WaitType],
         CAST ([W1].[WaitS] AS DECIMAL(14, 2)) AS [Wait_S],
         CAST ([W1].[ResourceS] AS DECIMAL(14, 2)) AS [Resource_S],
         CAST ([W1].[SignalS] AS DECIMAL(14, 2)) AS [Signal_S],
         [W1].[WaitCount] AS [WaitCount],
         CAST ([W1].[Percentage] AS DECIMAL(4, 2)) AS [Percentage],
         CAST (([W1].[WaitS] / [W1].[WaitCount]) AS DECIMAL (14, 4))
                                                    AS [AvgWait_S],
         CAST (([W1].[ResourceS] / [W1].[WaitCount]) AS DECIMAL (14, 4))
                                                     AS [AvgWait_S],
         CAST (([W1].[ResourceS] / [W1].[WaitCount]) AS DECIMAL (14, 4))
                                                     AS [AvgRes_S],
         CAST (([W1].[SignalS] / [W1].[WaitCount]) AS DECIMAL (14, 4))
                                                    AS [AvgSig_S]
      FROM [Waits] AS [W1]
      INNER JOIN [Waits] AS [W2]
         ON [W2].[RowNum] <= [W1].[RowNum]
      GROUP BY [W1].[RowNum], [W1].[wait_type], [W1].[WaitS],
         [W1].[ResourceS], [W1].[SignalS], [W1].[WaitCount],
          [W1].[Percentage]
      HAVING SUM ([W2].[Percentage]) - [W1].[Percentage] < 95;'
            );
GO
```

Listing 10: Capturing wait stats data for analysis

We should capture wait statistics regularly, at least once a week or once a month. We could do so more frequently, perhaps daily, but remember that unless we are clearing the data regularly, they represent an aggregation of waits since the last restart. The longer the waits have been accumulating, the harder it may be to spot smaller changes in wait percentages. For example, let's say the system suffers a short period (one hour) of poor performance, during which the number of, and duration of, a certain wait type increases significantly. This 'spike' may be hard to spot if you're analyzing waits accumulated over a long period (e.g. a month), since the spike might not affect significantly the overall wait percentage for that period.

Reviewing wait statistics data

We'll want to review regularly the waits for each SQL Server instance. If you capture them once a week, then check on the trending once a week. The simple `SELECT` in Listing 11 retrieves from the `dbo.WaitStats` table all the data captured for the last 30 days.

```
SELECT  *
FROM    [dbo].[WaitStats]
WHERE   [CaptureDate] > GETDATE() - 30
ORDER BY [RowNum];
```

Listing 11: Reviewing the last 30 days' data

If you need to view older data, adjust the number of days as necessary, or remove the predicate entirely. In some cases, it might be ideal to look at only the top wait for each set of data collected, as shown in Listing 12 (again, alter the number of days as needed).

```
SELECT  [w].[CaptureDate] ,
        [w].[WaitType] ,
        [w].[Percentage] ,
        [w].[Wait_S] ,
        [w].[WaitCount] ,
        [w].[AvgWait_S]
FROM    [dbo].[WaitStats] w
        JOIN ( SELECT   MIN([RowNum]) AS [RowNumber] ,
                        [CaptureDate]
               FROM     [dbo].[WaitStats]
               WHERE    [CaptureDate] IS NOT NULL
                        AND [CaptureDate] > GETDATE() - 30
               GROUP BY [CaptureDate]
             ) m ON [w].[RowNum] = [m].[RowNumber]
ORDER BY [w].[CaptureDate];
```

Listing 12: Reviewing the top wait for each collected data set

There are many ways in which to examine this data, but our focus, initially, should be to understand the top waits in our system and ensure they're consistent over time. Expect to tweak the capture and monitoring process in the first few weeks of implementation.

As discussed earlier, it's up to the DBAs to decide on a clear, consistent policy on how often to collect and analyze this data, and when to clear out the `sys.dm_os_wait_stats` DMV. Here, by way of a starting point, we offer three possible options for clearing, capturing, and reviewing this data:

Option 1:

- Never clear wait statistics
- Capture weekly (at the end of any business day)
- Review weekly

Option 2:

- Clear wait statistics on Sunday nights (or after a full weekly backup)
- Capture daily at the end of the business day
- Review daily, checking to see if the percentages for wait types vary throughout the week

Option 3:

- Clear wait statistics nightly (after full or differential backups complete)
- Capture daily, at the end of the business day (optional: capture after any evening or overnight processing)
- Review daily, checking to see how the waits and their percentages vary throughout the week (and throughout the day if capturing more than once a day)

The critical point to remember is that you are capturing this data to achieve a baseline, and to understand "normal" wait patterns on your systems. However, it's common, as you're reviewing this information, to identify existing or potential bottlenecks. This is a good thing. Having this information allows you to investigate an unexpected or high wait type, and determine the possible source of the bottleneck that caused the wait, before it becomes a production problem.

Managing historical data

As with all baseline data, it will cease to be relevant after a certain point, and you can remove it from the `BaselineData` database. The query in Listing 13 removes data older than 90 days, but you can adjust this value as appropriate for your environment. The overall size of the `dbo.WaitStats` table will depend on the choices you make on how often to capture the data and how long to retain it.

```
DELETE  FROM [dbo].[WaitStats]
WHERE   [CaptureDate] < GETDATE() – 90;
```

Listing 13: Purging data over 90 days old

Alternatively, you could implement a `dbo.usp_PurgeOldData` stored procedure, as shown in Listing 14.

```
IF OBJECTPROPERTY(OBJECT_ID(N'usp_PurgeOldData'), 'IsProcedure') = 1
     DROP PROCEDURE usp_PurgeOldData;
GO

CREATE PROCEDURE dbo.usp_PurgeOldData
    (
      @PurgeWaits SMALLINT

    )
AS
    BEGIN;
        IF @ PurgeWaits IS NULL
            BEGIN;
               RAISERROR(N'Input parameters cannot be NULL', 16, 1);
               RETURN;
            END;

        DELETE  FROM [dbo].[WaitStats]
        WHERE   [CaptureDate] < GETDATE() - @PurgeWaits;
    END;
```

Listing 14: The `dbo.usp_PurgeOldData` stored procedure

To wait statistics data older than 90 days, execute the following statement:

```
EXEC dbo.usp_PurgeOldData 90
```

Summary

We hope that this whitepaper offers a good "end-to-end" view of how to use wait statistics to troubleshoot current and historical performance issues on your SQL Server instances. Wait statistics are a very useful diagnostic tool that can help us direct our tuning efforts and avoid wasting time, but are most effective when we use them in conjunction with supporting tools and data, such as virtual files statistics and performance counters.

Further reading

1. Troubleshooting SQL Server: A Guide for the Accidental DBA (free eBook), by Jonathan Kehayias and Ted Krueger
https://www.simple-talk.com/books/sql-books/troubleshooting-sql-server-a-guide-for-the-accidental-dba/

2. Performance Tuning using SQL Server Dynamic Management Views (free eBook), by Louis Davidson and Tim Ford
https://www.simple-talk.com/books/sql-books/performance-tuning-with-sql-server-dynamic-management-views/

3. Wait statistics, or please tell me where it hurts (blog post), by Paul Randal
http://www.sqlskills.com/blogs/paul/wait-statistics-or-please-tell-me-where-it-hurts/

4. SQL Server Performance Troubleshooting Using Wait Statistics (Pluralsight video training course, subscription required), by Paul Randal
http://pluralsight.com/training/Courses/TableOfContents/sqlserver-waits

5. Diagnosing and Resolving Latch Contention on SQL Server (Microsoft whitepaper)
http://www.microsoft.com/en-us/download/details.aspx?id=26665

6. Trimming Transaction Log Fat (blog post), by Paul Randal
http://www.sqlperformance.com/2012/12/io-subsystem/trimming-t-log-fat

7. Description of the waittype and lastwaittype columns in the master.dbo.sysprocesses table in SQL Server 2000 and SQL Server 2005 (Microsoft Support article)
http://support.microsoft.com/kb/822101

About the authors

Erin Stellato is a Principal Consultant with SQLskills. Erin discovered a passion for all things data while completing her M.S. in Motor Control at the University of Michigan. As she entered the IT field, she discovered the power of relational databases. During her IT career, Erin has worked closely with customers, users and colleagues to support business solutions across a variety of industries including healthcare, insurance, finance and government. These experiences provided opportunities to see varied implementations, troubleshoot hardware-related problems, give guidance on configuration and design, and provide consultation on optimization, availability and recoverability. Not surprisingly, Erin's interests include Internals, Performance Tuning, High Availability and Disaster Recovery. More than anything though, she likes to know how SQL Server works, so she can fix it when it breaks.

Erin is an active member of the SQL Server community. She currently runs the PASS Performance Virtual Chapter and is involved in the Ohio North SQL Server User Group. She also presents at user groups, SQLSaturdays and the PASS Summit, and blogs about her experiences at SQLskills.

Jonathan Kehayias is a Principal Consultant and Trainer for SQLskills, one of the most well-known and respected SQL Server training and consulting companies in the world. Jonathan is a SQL Server MVP and one of the few Microsoft Certified Masters for SQL Server 2008 outside of Microsoft. Jonathan frequently blogs about SQL Server, presents sessions at PASS Summit, SQLBits, SQL Connections and local SQL Saturday events, and has remained a top answerer of questions on the MSDN SQL Server Database Engine forum since 2007. He is also author of the book Troubleshooting SQL Server: A Guide for the Accidental DBA.

Jonathan is a performance tuning expert, for both SQL Server and hardware, and has architected complex systems as a developer, business analyst, and DBA. Jonathan also has extensive development (T-SQL, C#, and ASP.Net), hardware and virtualization design expertise, Windows expertise, Active Directory experience, and IIS administration experience.

Outside of SQL Server, Jonathan is also a Drill Sergeant in the U.S. Army Reserves and is married with two young children. On most nights he can be found at the playground, in a swimming pool, or at the beach with his kids.

About the technical editor

Tony Davis is a highly-experienced technical editor at Red Gate Software, based in Cambridge (UK). He specializes in databases, and especially SQL Server. He edits and writes for both the Simple-talk.com and SQLServerCentral.com websites and newsletters, with a combined audience of over 1.5 million subscribers. You can sample his short-form writing at either his Simple-Talk.com blog or his SQLServerCentral.com author page.

As the editor behind most of the SQL Server books published by Red Gate, he spends much of his time helping others express what they know about SQL Server, but is also the lead author of the book, SQL Server Transaction Log Management.

In his spare time, he enjoys running, football, contemporary fiction and real ale.