[http://technet.microsoft.com/en-us/library/cc966540.aspx#\_I/O\_Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx%23_I/O_Bottlenecks)

**I/O Bottlenecks**

SQL Server performance depends heavily on the I/O subsystem. Unless your database fits into physical memory, SQL Server constantly brings database pages in and out of the buffer pool. This generates substantial I/O traffic. Similarly, the log records need to be flushed to the disk before a transaction can be declared committed. And finally, SQL Server uses **tempdb** for various purposes such as to store intermediate results, to sort, to keep row versions and so on. So a good I/O subsystem is critical to the performance of SQL Server.

Access to log files is sequential except when a transaction needs to be rolled back while access to data files, including **tempdb**, is randomly accessed. So as a general rule, you should have log files on a separate physical disk than data files for better performance. The focus of this paper is not how to configure your I/O devices but to describe ways to identify if you have I/O bottleneck. Once an I/O bottleneck is identified, you may need to reconfigure your I/O subsystem.

If you have a slow I/O subsystem, your users may experience performance problems such as slow response times, and tasks that abort due to timeouts.

You can use the following performance counters to identify I/O bottlenecks. Note, these AVG values tend to be skewed (to the low side) if you have an infrequent collection interval. For example, it is hard to tell the nature of an I/O spike with 60-second snapshots. Also, you should not rely on one counter to determine a bottleneck; look for multiple counters to cross check the validity of your findings.

* **PhysicalDisk Object: Avg. Disk Queue Length** represents the average number of physical read and write requests that were queued on the selected physical disk during the sampling period. If your I/O system is overloaded, more read/write operations will be waiting. If your disk queue length frequently exceeds a value of 2 during peak usage of SQL Server, then you might have an I/O bottleneck.
* **Avg. Disk Sec/Read** is the average time, in seconds, of a read of data from the disk. Any number
* Less than 10 ms - very good
* Between 10 - 20 ms - okay
* Between 20 - 50 ms - slow, needs attention
* Greater than 50 ms – Serious I/O bottleneck
* **Avg. Disk Sec/Write** is the average time, in seconds, of a write of data to the disk. Please refer to the guideline in the previous bullet.
* **Physical Disk: %Disk Time** is the percentage of elapsed time that the selected disk drive was busy servicing read or write requests. A general guideline is that if this value is greater than 50 percent, it represents an I/O bottleneck.
* **Avg. Disk Reads/Sec** is the rate of read operations on the disk. You need to make sure that this number is less than 85 percent of the disk capacity. The disk access time increases exponentially beyond 85 percent capacity.
* **Avg. Disk Writes/Sec** is the rate of write operations on the disk. Make sure that this number is less than 85 percent of the disk capacity. The disk access time increases exponentially beyond 85 percent capacity.

When using above counters, you may need to adjust the values for RAID configurations using the following formulas.

Raid 0 -- I/Os per disk = (reads + writes) / number of disks

Raid 1 -- I/Os per disk = [reads + (2 \* writes)] / 2

Raid 5 -- I/Os per disk = [reads + (4 \* writes)] / number of disks

Raid 10 -- I/Os per disk = [reads + (2 \* writes)] / number of disks

For example, you have a RAID-1 system with two physical disks with the following values of the counters.

Disk Reads/sec            80

Disk Writes/sec           70

Avg. Disk Queue Length    5

In that case, you are encountering (80 + (2 \* 70))/2 = 110 I/Os per disk and your disk queue length = 5/2 = 2.5 which indicates a border line I/O bottleneck.

You can also identify I/O bottlenecks by examining the latch waits. These latch waits account for the physical I/O waits when a page is accessed for reading or writing and the page is not available in the buffer pool. When the page is not found in the buffer pool, an asynchronous I/O is posted and then the status of the I/O is checked. If I/O has already completed, the worker proceeds normally. Otherwise, it waits on PAGEIOLATCH\_EX or PAGEIOLATCH\_SH, depending upon the type of request. The following DMV query can be used to find I/O latch wait statistics.

Select  wait\_type,

        waiting\_tasks\_count,

        wait\_time\_ms

from    sys.dm\_os\_wait\_stats

where    wait\_type like 'PAGEIOLATCH%'

order by wait\_type

wait\_type       waiting\_tasks\_count  wait\_time\_ms   signal\_wait\_time\_ms

-----------------------------------------------------------------------

PAGEIOLATCH\_DT  0                    0                    0

PAGEIOLATCH\_EX  1230                 791                  11

PAGEIOLATCH\_KP  0                    0                    0

PAGEIOLATCH\_NL  0                    0                    0

PAGEIOLATCH\_SH  13756                7241                 180

PAGEIOLATCH\_UP  80                   66                   0

Here the latch waits of interest are the underlined ones. When the I/O completes, the worker is placed in the runnable queue. The time between I/O completions until the time the worker is actually scheduled is accounted under the signal\_wait\_time\_ms column. You can identify an I/O problem if your waiting\_task\_counts and wait\_time\_ms deviate significantly from what you see normally. For this, it is important to get a baseline of performance counters and key DMV query outputs when SQL Server is running smoothly. These wait\_types can indicate whether your I/O subsystem is experiencing a bottleneck, but they do not provide any visibility on the physical disk(s) that are experiencing the problem.

You can use the following DMV query to find currently pending I/O requests. You can execute this query periodically to check the health of I/O subsystem and to isolate physical disk(s) that are involved in the I/O bottlenecks.

select

    database\_id,

    file\_id,

    io\_stall,

    io\_pending\_ms\_ticks,

    scheduler\_address

from    sys.dm\_io\_virtual\_file\_stats(NULL, NULL)t1,

        sys.dm\_io\_pending\_io\_requests as t2

where    t1.file\_handle = t2.io\_handle

A sample output is as follows. It shows that on a given database, there are three pending I/Os at this moment. You can use the database\_id and file\_id to find the physical disk the files are mapped to. The io\_pending\_ms\_ticks represent the total time individual I/Os are waiting in the pending queue.

Database\_id File\_Id io\_stall io\_pending\_ms\_ticks scheduler\_address

----------------------------------------------------------------------

6           1        10804        78            0x0227A040

6           1        10804        78            0x0227A040

6           2        101451       31            0x02720040

**Resolution**

When you have identified an I/O bottleneck, you can address it by doing one or more of the following:

* Check the memory configuration of SQL Server. If SQL Server has been configured with insufficient memory, it will incur more I/O overhead. You can examine following counters to identify memory pressure
  + Buffer Cache hit ratio
  + Page Life Expectancy
  + Checkpoint pages/sec
  + Lazywrites/sec

For more information on the memory pressure, see [Memory Bottlenecks](http://technet.microsoft.com/en-us/library/cc966540.aspx#_Memory_Bottlenecks).

* Increase I/O bandwidth.
  + Add more physical drives to the current disk arrays and/or replace your current disks with faster drives. This helps to boost both read and write access times. But don't add more drives to the array than your I/O controller can support.
  + Add faster or additional I/O controllers. Consider adding more cache (if possible) to your current controllers.
* Examine execution plans and see which plans lead to more I/O being consume. It is possible that a better plan (for example, index) can minimize I/O. If there are missing indexes, you may want to run Database Engine Tuning Advisor to find missing indexes

The following DMV query can be used to find which batches/requests are generating the most I/O. You will notice that we are not accounting for physical writes. This is ok if you consider how databases work. The DML/DDL statements within a request do not directly write data pages to disk. Instead, the physical writes of pages to disks is triggered by statements only by committing transactions. Usually physical writes are done by either by Checkpoint or by the SQL Server lazy writer. A DMV query like the following can be used to find the top five requests that generate the most I/Os. Tuning those queries so that they perform fewer logical reads can relieve pressure on the buffer pool. This allows other requests to find the necessary data in the buffer pool in repeated executions (instead of performing physical I/O). Hence, overall system performance is improved.

select top 5

    (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,

     Execution\_count,

    statement\_start\_offset as stmt\_start\_offset,

    sql\_handle,

    plan\_handle

from sys.dm\_exec\_query\_stats

order by

(total\_logical\_reads + total\_logical\_writes) Desc

You can, of course, change this query to get different views on the data. For example, to generate the top five requests that generate most I/Os in single execution, you can order by:

    (total\_logical\_reads + total\_logical\_writes)/execution\_count

Alternatively, you may want to order by physical I/Os and so on. However, logical read/write numbers are very helpful in determining whether or not the plan chosen by the query is optimal. For example, it may be doing a table scan instead of using an index. Some queries, such as those that use nested loop joins may have high logical counters but be more cache-friendly since they revisit the same pages.

**Example**: Let us take the following two batches consisting of two SQL queries where each table has 1000 rows and rowsize > 8000 (one row per page).

Batch-1

select

    c1,

    c5

from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4

order by c2

Batch-2

select \* from t1

For the purpose of this example, before running the DMV query, we clear the buffer pool and the procedure cache by running the following commands.

checkpoint

dbcc freeproccache

dbcc dropcleanbuffers

Here is the output of the DMV query. You will notice two rows representing the two batches.

Avg\_logical\_reads Avg\_logical\_writes Avg\_phys\_reads Execution\_count

stmt\_start\_offset

-----------------------------------------------------------------------

---------------

2794                1                385                1

    0

1005                0                0                  1

    146

sql\_handle                                         plan\_handle

-----------------------------------------------------------------------

-----

0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB

x0600050099EC8520A8619803000000000000000000000000

0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB

x0600050099EC8520A8619803000000000000000000000000

You will notice that the second batch only incurs logical reads but no physical I/O. This is because the data it needs was already cached by the first query (assuming there was sufficient memory).

You can get the text of the query by running the following query.

select text

from sys.dm\_exec\_sql\_text(

     0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB)

Here is the output.

select

    c1,

    c5

from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4

order by c2

You can also find out the string for the individual statement by executing the following:

select

    substring(text,

              (<statement\_start\_offset>/2),

              (<statement\_end\_offset> -<statement\_start\_offset>)/2)

from sys.dm\_exec\_sql\_text

(0x0200000099EC8520EFB222CEBF59A72B9BDF4DBEFAE2B6BB)

The value of statement\_start\_offest and statement\_end\_offset need to be divided by two in order to compensate for the fact that SQL Server stores this kind of data in Unicode. A statement\_end\_offset value of -1, indicates that the statement does go up to the end of the batch. However the **substring()** function does not accommodate -1 as a valid value. Instead of using -1 as (<statement\_end\_offset> -<statement\_start\_offset>)/2, one should enter the value 64000, which should make sure that the statement is covered in all cases. With this method, a long-running or resource-consuming statement can be filtered out of a large stored procedure or batch.

Similarly, you can run the following query to find to the query plan to identify if the large number of I/Os is a result of a poor plan choice.

select \*

from sys.dm\_exec\_query\_plan

    (0x0600050099EC8520A8619803000000000000000000000000)