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 storing intermediate results, sorting, and keeping row versions. 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 data files, including **tempdb**, are randomly accessed. So as a general rule, you should have log files on a physical disk that is separate from the 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 whether you have I/O bottleneck. After 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 do not complete due to time-outs.

You can use the following performance counters to identify I/O bottlenecks. Note that 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, 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. The following list shows ranges of possible values and what the ranges mean:

- 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. The guidelines for the **Avg. Disk Sec/Read** values apply here.

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, there is an I/O bottleneck.

Avg. Disk Reads/Sec is the rate of read operations on the disk. Ensure 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. Ensure that this number is less than 85 percent of the disk capacity. The disk access time increases exponentially beyond 85 percent capacity.

When you use these 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 might 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 borderline 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 the I/O has already completed, the worker proceeds normally. Otherwise, it waits on PAGEIOLATCH_EX or PAGEIOLATCH_SH, depending upon the type of request. You can use the following DMV query 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
```

A sample output follows.

```
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
```

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 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** columns to find the physical disk the files are mapped to. The **io_pending_ms_ticks** values 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 see an I/O bottleneck, your first instinct might be to upgrade the I/O subsystem to meet the workload requirements. This will definitely help, but before you go out and invest money in hardware, examine the I/O bottleneck to see whether it is the result of poor configuration and/or query plans. We recommend you to follow the steps below in strict order.

1. **Configuration:** 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 the following counters to identify memory pressure:
    - Buffer Cache hit ratio
    - Page Life Expectancy
    - Checkpoint pages/sec
    - Lazywrites/sec

For more information about memory pressure, see Memory Bottlenecks earlier in this paper.

2. **Query Plans**: Examine execution plans and see which plans lead to more I/O being consumed. 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 or requests are generating the most I/O. Note that we are not accounting for physical writes. This is okay if you consider how databases work. The DML and 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 either by checkpoint or by the SQL Server lazy writer. You can use a DMV query like the following to find the 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 enables 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.

   Here is an example of a query that joins two tables with a hash join.

```
create table t1 (c1 int primary key, c2 int, c3 char(8000))

create table t2  (C4 int, c5 char(8000))

go


--load the data

declare @i int

select @i = 0

while (@i < 6000)

begin

    insert into t1 values (@i, @i + 1000, 'hello')

   insert into t2 values (@i,'there')

   set @i = @i + 1

end
--now run the following query
select c1, c5

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

order by c2
```

Run another query so that there are two queries to look at for I/O stats

```
select SUM(c1) from t1
```

These two queries are run in the single batch. Next, use the following DMV query to examine the queries that generate the most I/Os

```
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,
    (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(MAX),text)) * 2
                ELSE statement_end_offset
            END - statement_start_offset)/2)
     FROM sys.dm_exec_sql_text(sql_handle)) AS query_text,
      (SELECT query_plan from sys.dm_exec_query_plan(plan_handle)) as
query_plan
FROM sys.dm_exec_query_stats
   ORDER BY (total_logical_reads +
   total_logical_writes)/execution_count DESC
```

You can, of course, change this query to get different views on the data. For example, to generate the five requests that generate the 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. The output of the query is as follows.

```
avg_logical_reads    avg_logical_writes   avg_phys_reads
-----------------    ------------------   --------------
16639                10                   1098
6023                 0                    0
execution_count      stmt_start_offset
---------------      -----------------
1                    0
1                    154

Query_text                                  Query_plan
----------------------------------          -----------
select c1, c5  from t1 INNER HASH JOIN …    <link to query plan>
select SUM(c1) from t1                      <link to query plan>
```

The output tells you several important things. First, it identifies the queries that are generating the most I/Os. You can also look at the SQL text to see whether the query needs to be re-examined to reduce I/Os. Verify that the query plan is optimal. For example, a new index might be helpful. Second, the second query in the batch does not incur any physical I/Os because all the pages needed for table t1 are already in the buffer pool. Third, the execution count can be used to identify whether it is a one-off query or the one that is executed frequently and therefore needs to be looked into carefully.

3. **Data Compression:** Starting with SQL Server 2008, you can use the data compression feature to reduce the size of tables and indexes, thereby reducing the size of the whole database. The compression achieved depends on the schema and the data distribution. Typically, you can achieve 50-60% compression. We have seen up to 90% compression in some cases. What it means to you is that if you are able to compress you active data 50%, you have in effect reduced your I/O requirements by half. Data compression comes at the cost of additional CPU, which needs to be weighed in for your workload. Here are some general strategies.

   First, why isn't compressing the whole database blindly such a good idea? Well, to give you an extreme example, if you have a heavily used table T with 10 pages in a database with millions of pages, there is no benefit in compressing T. Even if SQL Server could compress 10 pages to 1 page, you hardly made a dent in the size of the database, but you did add some CPU overhead instead. In a real-life workload, the choices are not this obvious, but this example shows that you must look before you compress. Our recommendation is this: Before you compress an object (for example, a table index or a partition), look at its size, usage, and estimated compression savings by using the **sp_estimate_data_compression_savings** stored procedure.

   Let us look at each of these in some detail:

- If the size of the object is much smaller than the overall size of the database, it does not buy you much.
- If the object is used heavily both for DML and SELECT operations, you will incur additional CPU overhead that can impact your workload, especially if it makes it CPU bound. You can use **sys.dm_db_index _operational_stats** to find the usage pattern of objects to identify which tables, indexes, and partitions are being hit the most.
- The compression savings are schema-dependent and data-dependent, and in fact, for some objects, the size after compression can be larger than before, or the space savings can be insignificant.

If you have a partitioned table where data in some partitions is accessed infrequently, you may want to compress those partitions and associated indexes with page compression. This is a common scenario with partitioned tables where older partitions are referenced infrequently. For example, you might have a table in which sales data is partitioned by quarters across many years. Commonly the queries are run on the current quarter; data from other quarters is not referenced as frequently. So when the current quarter ends, you can change the compression setting for that quarter's partition.

4. **Upgrading the I/O Subsystem**: If you have confirmed that SQL Server is configured correctly and examined the query plans and you are still experiencing I/O bottlenecks, the last option left is to upgrade your I/O subsystem to 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.