Baselining with SQL Server Dynamic Management Views

02 July 2013
by Louis Davidson

When you're monitoring SQL Server, it is better to capture a baseline for those aspects that you're checking, such as workload, Physical I/O or performance. Once you know what is normal, then performance tuning and resource provisioning can be done in a timely manner before any problem becomes apparent. We can prevent problems by being able to predict them. Louis shows how to get started.

With the advent of the Dynamic Management Views (DMVs) in SQL Server 2005, Microsoft vastly expanded the range and depth of metadata that could be exposed regarding the connections, sessions, transactions, statements and processes that are, or have been, executing against a database instance. These DMOs provide insight into the resultant workload generated on the server, where the pressure points are, and so on, and are a significant and valuable addition to the DBA's troubleshooting armory.

In short, if you look hard enough, you will find an almost overwhelming amount of data regarding user activity on your SQL Server instances, and use and abuse of the available CPU, I/O and memory. Unfortunately, this data is often useless without proper context. It is very difficult to pinpoint a problem just by looking at a single, point-in-time snapshot of the data, or to spot retrospectively what caused a problem by examining aggregated data for the server, collected over many months.

This is why we need to capture **baselines** for this data, so that when attempting to diagnose a particular problem they know what "normal" looks like. It's an activity many DBAs end up postponing indefinitely perhaps because it seems like a huge task to work out which data to collect, let alone to start collect it on a regular basis, manage its storage over time, and perform detailed comparative analysis.

I hope that this article will prove that getting started with baselines isn't as hard as you might think. You've probably already established some troubleshooting queries of the type `SELECT Value FROM SomeSystemTable`. Capturing a set of baseline values for such a query can be as easy as changing it as follows:

```
INSERT into BaseLine.SomeSystemTable (value, captureTime) SELECT Value, SYSDATETIME() FROM
SomeSystemTable;
```

I'll demonstrate this with a few examples of how to capture baselines from DMV data, specifically how to capture baselines to:

- Measure I/O distribution across your database files
- Track "per second average" PerfMon counter values, such as "Page Lookups per second"

Point in time versus cumulative DMV data

We can query data held on the DMVs just as we would any other table, view or function. However, always remember that the data returned is "dynamic" in nature. SQL Server collects it from a range of different structures in the database engine and it represents, in the main, a point-in-time "snapshot" of the activity that was occurring on the server at the time we executed the DMV query.

Sometimes, this is exactly what is required; we have a performance issue right now, and want to find out which sessions and queries are running, right now. Bear in mind, though, that this point-in-time data can and likely will change each time you query it, as the state of the server changes. Occasionally, you should expect to see anomalous or non-representative results and you may need to run a script many times to get a true picture of the activity on your instance.

We need to capture baselines for this data, as it is quite difficult to query the data in these point-in-time DMOs in the hope that the problem will simply reveal itself. How many DBAs, for example, can run a query of the form *select [columns] from [locking DMV]*and know instantly whether the locking pattern revealed is normal?

In other cases, the data in a DMV will be cumulative. In other words, the data in a given column is accumulative and incremented every time a certain event occurs. In most cases, the data resets only when SQL Server restarts. If you have regular maintenance windows, for example to apply patches, then you can gauge from this the collection period.

In two cases (`sys.dm_os_latch_stats` and `sys.dm_os_wait_stats`), the data accumulates since server restart but we can also clear it out manually. For example, every time a session waits a period for a resource to become available, SQL Server records this in a column of the `sys.dm_os_wait_stats` DMV. When querying such a DMV, you will see the total amount of time spent waiting for various resources, across all sessions, since the server last restarted or someone cleared out the statistics manually, using `DBCCSQLPERF("sys.dm_os_wait_stats",CLEAR);`.

In all cases, when analyzing data accumulated over a long period, it will be hard to see the smaller details. You will want to capture a baseline (for example, a suitable period after a server restart, in order to capture the full workload), and then recapture the data at regular intervals thereafter. Likewise, if you want to measure the impact of a certain change to the database (a new index for example), you'll need to take a baseline measurement, make the change, and then measure the difference.

Baselining Physical I/O statistics

For each database file that SQL Server uses, which includes not only the data files, but also the log and full text files, the `sys.dm_io_virtual_file_stats` Dynamic Management Function provides cumulative physical I/O statistics, indicating how frequently the file has been used by the database for reads and writes, since the server was last rebooted. It also provides a very useful metric in the form of the "I/O stall" time, which indicates the total amount of time that user processes have waited for I/O to complete, on the file in question. Note that this object measures physical I/O only. Logical IO operations that read from cached data will not show up here.

To get an accurate view of the data, you'd need to reboot the server at same time every day, and take a snapshot at the same time so you could compare day-to-day activity and trends. Since this is neither practical nor a very good idea, we can, instead, take a 'baseline' measurement followed by the actual measurement and then subtract the two, so that you can see where I/O is "accumulating". In order to measure a "time slice" of activity, we take a baseline measurement, inserting data into a temporary table, as shown in Listing 1.

```
SELECT DB_NAME(mf.database_id) AS databaseName ,
    mf.physical_name ,
    divfs.num_of_reads ,
    divfs.num_of_bytes_read ,
    divfs.io_stall_read_ms ,
    divfs.num_of_writes ,
    divfs.num_of_bytes_written ,
    divfs.io_stall_write_ms ,
    divfs.io_stall ,
    size_on_disk_bytes ,
    GETDATE() AS baselineDate
INTO #baseline
FROM sys.dm_io_virtual_file_stats(NULL, NULL) AS divfs
    JOIN sys.master_files AS mf ON mf.database_id = divfs.database_id
          AND mf.file_id = divfs.file_id
```

Listing 1: Capturing baseline disk I/O statistics from `sys.dm_io_virtual_file_stats` in a temporary table

Listing 2 shows a query against the `#baseline` table, returning read statistics for a particular database.

```
SELECT physical_name ,
    num_of_reads ,
    num_of_bytes_read ,
    io_stall_read_ms
FROM #baseline
WHERE databaseName = 'DatabaseName'
```

Listing 2: Querying the `#baseline` temporary table

This returns the following data for the read statistics:

```
physical_name num_of_reads num_of_bytes_read
io_stall_read_ms
---------------------- -------------------- -------------------- --------
```

```
F:\MSSQ...DATABASE.mdf 1560418              381784449024         176090340
E:\MSSQ...BASE_log.LDF 925                  592683008            7000
I:\MSSQ...SE_index.ndf 398504               310491209728         39664904
k:\mssq...TABASE2A.mdf 540176               155267350528         319640508
```

This data, taken on a server that restarted about 12 hours previously, is not especially interesting or meaningful, in its own right. However, the next step is where turn this broadly interesting data into specific information that can pinpoint a problem that occurred at a specific time.

Having captured the baseline, wait a set amount of time, or for some process to complete, and then take a second measurement. On a very busy server, you may wait only 10 seconds before taking the second measurement, as is the case in this example. We use a CTE to capture the current values, then join it to the temporary table to subtract the baseline values and calcualte the difference in the readings.

```
WITH currentLine
     AS ( SELECT DB_NAME(mf.database_id) AS databaseName ,
               mf.physical_name ,
               num_of_reads ,
               num_of_bytes_read ,
               io_stall_read_ms ,
               num_of_writes ,
               num_of_bytes_written ,
               io_stall_write_ms ,
               io_stall ,
               size_on_disk_bytes ,
               GETDATE() AS currentlineDate
     FROM sys.dm_io_virtual_file_stats(NULL, NULL) AS divfs
               JOIN sys.master_files AS mf
                  ON mf.database_id = divfs.database_id
                     AND mf.file_id = divfs.file_id
     )
  SELECT currentLine.databaseName ,
     LEFT(currentLine.physical_name, 1) AS drive ,
     currentLine.physical_name ,
     --gets the time diference in milliseconds since
     -- the baseline was taken
     DATEDIFF(millisecond,baseLineDate,currentLineDate) AS elapsed_ms,
       currentLine.io_stall - #baseline.io_stall AS io_stall_ms ,
       currentLine.io_stall_read_ms - #baseline.io_stall_read_ms
                                  AS io_stall_read_ms ,
       currentLine.io_stall_write_ms - #baseline.io_stall_write_ms
                                  AS io_stall_write_ms ,
       currentLine.num_of_reads - #baseline.num_of_reads
                                  AS num_of_reads ,
       currentLine.num_of_bytes_read - #baseline.num_of_bytes_read
                                  AS num_of_bytes_read ,
       currentLine.num_of_writes - #baseline.num_of_writes
                                  AS num_of_writes ,
       currentLine.num_of_bytes_written - #baseline.num_of_bytes_written
                                  AS num_of_bytes_written
  FROM currentLine
     INNER JOIN #baseline
        ON #baseLine.databaseName = currentLine.databaseName
     AND #baseLine.physical_name = currentLine.physical_name
  WHERE #baseline.databaseName = 'DatabaseName'
```

Listing 3: Capturing 10 seconds of disk I/O statistics, since the baseline measurement

Following are a sampling of the result, again focusing only on the read statistics:

```
physical_name  elapsed_ms num_of_reads num_of_bytes_read  io_stall_read_ms
-------------- ---------- ------------ ------------------ ------------------
```

```
F:\MSSQ.SE.mdf   10016         915              128311296         34612
E:\MSSQ.og.LDF   10016           0                      0             0
I:\MSSQ.ex.ndf   10016         344              172933120          8000
k:\mssq.2A.mdf   10016           0                      0             0
```

These results show that over the 10-second sampling period, processes using the data file on the F: drive waited a combined total of 34 seconds. Of course, this data would have to be assessed in light of how many processes ran during the sampling period; if it was four, then the result would be very worrying, if it was 100 then perhaps less so.

It is interesting that user processes read more data from the I: drive, with fewer I/O stalls, which is most likely explained by different usage patterns. The I: drive is subject to a smaller number of mostly sequential reads, whereas the F: drive is subject to many more reads that are mainly random in nature. Obviously, one can only know for sure with some with some knowledge of the activity that was occurring during the sampling. In any event, it is certainly worrying to see that, on the F: drive, the stall times are substantially greater than the elapsed time, and I'd want to investigate further to find out why.

I hope that this simple example illustrates how, by taking baselines, we can analyze this data to identify file I/O bottlenecks. By comparing this data to that obtained routinely from performance counters, profiler traces and other DMV snapshots, we can really start to get an idea of how our disks are being utilized for a given server workload.

We can expand this technique to a more permanent solution by storing the rows in a permanent table, giving you the ability to do the math on rows to see what is happening, and what has happened – see my What Counts for a DBA webinar for more detailed coverage of this. The only caveat is that you have to deal with restarts and resets of the DMV data, which is a small problem at most.

Baselining PerfMon Data using the DMVs

SQL Server provides a number of database-level and instance-level objects and associated counters, which we can use to monitor various aspects of SQL Server performance. SQL Server exposes these counters in the `sys.dm_os_performance_counters` DMV. These counters indicate the "queues" in your system; the places where there is a lot of demand for a given resource, and the reasons for the excessive demand, via specific resource measurements such as disk writes/sec, processor queue lengths, available memory, and so on.

Generally, these performance counters are investigated using Performance Monitor (PerfMon), a Windows OS monitoring tool that provides a vast range of counters for monitoring memory, disk, CPU and network usage on a server (for example, see http://technet.microsoft.com/en-us/library/cc768048.aspx), and also exposes the counters maintained by SQL Server. Most DBAs will be familiar with setting up PerfMon to record statistics from various counters at regular intervals, storing the data in a file and then importing it into Excel for analysis.

However if, like me, you prefer to save the statistics in a database table and interrogate them using SQL, the `sys.dm_os_performance_counters` DMV is a very useful tool. Just write the query to retrieve the data from the DMV, add `INSERTINTO CounterTrendingTableName…` and you have a rudimentary monitoring system! In addition, it's not always possible to get direct access to PerfMon, and accessing it from a different machine can be slow.

Unfortunately, using this DMV is far from plain sailing. With that warning in mind, let's take a look at the columns that the `sys.dm_os_performance_counters` DMV provides.

- **`object_name`** – name of the object to which the counter refers. This is usually a two-part name, starting with `SQL Server:`. For example, `SQL Server:Databases` or `SQL Server:Locks`.
- **`counter_name`** – name of the counter. For example, the `SQL Server:Databases` object exposes the `Log Shrinks` counter, to monitor transaction log shrink events.
- **`instance_name`** – specific instance of a counter, such as the database name for `SQLServer:Databases:LogShrinks` or users errors for `SQLServer:SQL Errors:Errors/sec`.
- **`cntr_value`** – most recent value of the counter.
- **`cntr_type`** – type of counter.

Note that only SQL Server counters are represented in the DMV, not any Windows or other counters.

Most of these columns look innocuous enough, but don't be deceived – the `cntr_value` and the `cntr_type` values, in particular, are a nest of vipers. The `cntr_type` column exposes WMI Performance Counter Types and, for the most part, the values provided for each type of counter in `cntr_value` will need to be decoded before we can use them. To get the list of counter types, we can execute the query shown in Listing 4.

```
SELECT DISTINCT
   cntr_type
FROM sys.dm_os_performance_counters
ORDER BY cntr_type

cntr_type
-----------
65792
272696576
537003264
1073874176
1073939712
```

Listing 4: Returning a list of PerfMon counter types.

Books Online does a poor job of documenting the `cntr_type` values, but my research revealed the following:

- **`65792 = PERF_COUNTER_LARGE_RAWCOUNT`** – provides the last observed value for the counter; for this type of counter, the values in `cntr_value` can be used directly, making this the most easily usable type
- **`272696576 = PERF_COUNTER_BULK_COUNT`** – provides the average number of operations per second. Two readings of `cntr_value` will be required for this counter type, in order to get the per second averages
- **`537003264 = PERF_LARGE_RAW_FRACTION`** – used in conjunction with `PERF_LARGE_RAW_BASE` to calculate ratio values, such as the cache hit ratio.
- **`1073874176 = PERF_AVERAGE_BULK`** – used to calculate an average number of operations completed during a time interval; like `PERF_LARG_RAW_FRACTION`, it uses `PERF_LARGE_RAW_BASE` to do the calculation
- **`1073939712 = PERF_LARGE_RAW_BASE`**, used in the translation of `PERF_LARGE_RAW_FRACTION` and `PERF_AVERAGE_BULK` values to readable output; should not be displayed alone.

Here, we'll focus only on the "Per second average" (`PERF_COUNTER_BULK_COUNT`) category of Perfmon Counters. Some interesting values to look out for in this category include:

- **`Server:Buffer Manager-Page lookups/sec`**
  gives an indication of cache activity; higher numbers indicate a more active buffer pool (the definition of "higher" is largely dependent on your hardware and usage)
- **`Server:Databases-<databaseName>- Log Bytes Flushed/sec`**
  gives an indication of how much data has been written to the log for the database
- **`SQLServer:Locks-_Total - Lock Requests/sec`**
  provides the number of locks being taken on a server per second, usually to compare to other time periods, to see when the server is being inundated with queries that might block other users.

As "point-in-time" values, a single snapshot does not necessarily tell us very much. However, when tracked over time, they can help us identify worrying events, trends, or changes. For example, let's say that users start experiencing performance problems at a given time of day, and that you notice that this coincides with spikes in the number of lock requests per second, which, in turn, coincides with the time that Joe Doofus, the manager with more rights than brains, issues a major query in `SERIALIZABLE` isolation level.

Nearly all of the counters in this group are named with a suffix of '/sec', but a few are actually prefixed '(ms)'. However, we can interpret the latter as "number of milliseconds waited per second." So, for example, `Total Latch Wait Time (ms)`' is the average amount of time per second that a certain process had to wait to acquire a latch, over the time the sample was taken. These counters are constantly incrementing values, though if they ever hit the maximum value, they would reset to zero.

The way to deal with these counters is the same as for our example with an accumulating counter DMV. Take a baseline value, wait some number of seconds, then sample again. For ad-hoc monitoring of a given operation, you would set the delay such that you could capture a suitable number of samples over the period of time the operation is taking place.

The example in Listing 5 uses a simple `WAITFOR` statement to implement the delay, in this case, 5 seconds. It uses a `datetime` column, with a default of `getdate()`, in order to capture the exact time the values were sampled (since the delay may not be exactly 5 seconds each time; for example, it might actually take 5020 milliseconds to execute the query).

```
DECLARE @PERF_COUNTER_BULK_COUNT INT
SELECT @PERF_COUNTER_BULK_COUNT = 272696576

--Holds initial state
DECLARE @baseline TABLE
   (
      object_name NVARCHAR(256) ,
      counter_name NVARCHAR(256) ,
      instance_name NVARCHAR(256) ,
      cntr_value BIGINT ,
      cntr_type INT ,
      time DATETIME DEFAULT ( GETDATE() )
   )

DECLARE @current TABLE
   (
      object_name NVARCHAR(256) ,
      counter_name NVARCHAR(256) ,
      instance_name NVARCHAR(256) ,
      cntr_value BIGINT ,
      cntr_type INT ,
      time DATETIME DEFAULT ( GETDATE() )
   )

--capture the initial state of bulk counters
INSERT INTO @baseline
   ( object_name ,
     counter_name ,
     instance_name ,
     cntr_value ,
     cntr_type
   )
   SELECT object_name ,
          counter_name ,
          instance_name ,
          cntr_value ,
          cntr_type
   FROM sys.dm_os_performance_counters AS dopc
   WHERE cntr_type = @PERF_COUNTER_BULK_COUNT

WAITFOR DELAY '00:00:05' --the code will work regardless of delay chosen

--get the followon state of the counters
INSERT INTO @current
   ( object_name ,
     counter_name ,
     instance_name ,
     cntr_value ,
     cntr_type
   )
   SELECT object_name ,
          counter_name ,
          instance_name ,
          cntr_value ,
          cntr_type
   FROM sys.dm_os_performance_counters AS dopc
   WHERE cntr_type = @PERF_COUNTER_BULK_COUNT

SELECT dopc.object_name ,
       dopc.instance_name ,
       dopc.counter_name ,
       --ms to second conversion factor
       1000 *
```

```
            --current value less the previous value
        ( ( dopc.cntr_value - prev_dopc.cntr_value )
            --divided by the number of milliseconds that pass
            --casted as float to get fractional results. Float
            --lets really big or really small numbers to work
            / CAST(DATEDIFF(ms, prev_dopc.time, dopc.time) AS FLOAT) )
                                                AS cntr_value
            --simply join on the names of the counters
FROM @current AS dopc
     JOIN @baseline AS prev_dopc ON prev_dopc. object_name =
dopc. object_name
                       AND prev_dopc.instance_name = dopc.instance_name
                       AND prev_dopc.counter_name = dopc.counter_name
WHERE dopc.cntr_type = @PERF_COUNTER_BULK_COUNT
      AND 1000 * ( ( dopc.cntr_value - prev_dopc.cntr_value )
                 /  CAST( DATEDIFF(ms, prev_dopc. time, dopc. time)  AS FLOAT) )
 /* default to only showing non-zero values */ <> 0
ORDER BY dopc. object_name ,
         dopc.instance_name ,
         dopc.counter_name
```

Listing 5: Returning the values of "per second average" PerfMon counters.

It is easy to adapt this code to another specific counter type, such as one of those listed previously but I won't show it here. Notice that we default to only showing non-zero values, since that is usually what you will be interested in when using this set of data.

Summary

Any monitoring tool, whether it takes the form of a set of hand-rolled scripts and some Agent jobs, or a proper third-party monitoring tool, will track and interrogate the data in various system tables and views over time. Armed with this monitoring data, performance tuning, and resource provisioning, starts to become more proactive than reactive. We can see problems coming and prevent escalation, rather than simply react after the event. We can create some Agent jobs to warn us as soon as a metric strays significantly from a value that represents "normal" for our system, as established by our baseline measurements.

Of course, there is quite a bit of data to capture, manage and monitor over time and what I've showed here is really just the start. For more ideas, check out Erin Stellato's series over on SQLServerCentral (http://www.sqlservercentral.com/Authors/Articles/Erin_Stellato/351331/).