Using Management Data Warehouse for Performance Monitoring

Microsoft Corporation
Published: July 2010

Author: Ken Lassesen

Abstract
This white paper provides best practices for performance management of SQL Server 2008 databases. Implementing these best practices can help you avoid or minimize common problems and optimize the performance of SQL Server so that you can effectively manage your resources, reduce operating expenses, increase productivity, and improve employee satisfaction.
Copyright Information

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, SQL Server, Hyper-V, MSDN, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

About the Author

Ken Lassesen has been involved with SQL Server performance testing since the first betas of SQL Server. Ken has a Master of Science in Commerce with a focus on Operations Research, Information Management and Statistics. Microsoft’s Internal Technology Group gave him the nickname of “Dr. Science” because of the skills he brought to both building a stressing toolkit and analyzing the results for some of the world’s largest SQL Server databases. Ken is also known as being part of the original Dr.GUI team for MSDN.

Ken is semi-retired and enjoys frequent hikes in the North Cascade Mountains.
# Table of Contents

**OVERVIEW** ............................................................................................................................................... 5

**WHAT IS A MANAGEMENT DATA WAREHOUSE** ....................................................................................... 5

  - Management Data Warehouse .............................................................................................................. 6
  - Activating MDW ................................................................................................................................. 7
  - Built-in Reports ............................................................................................................................... 16
  - Advanced Troubleshooting and Analysis ......................................................................................... 19
  - Modifying MDW for Long Term Tracking ....................................................................................... 19

**EXTENDING MDW** .................................................................................................................................. 23

  - Creating a Custom Data Collector ................................................................................................. 25
  - Starting and Stopping Custom Data Collection ............................................................................... 28
  - Custom Data Collection Tables ...................................................................................................... 30
  - Trouble Shooting ............................................................................................................................. 31

**RETRIEVING DATA FROM MDW FOR ANALYSIS** ...................................................................................... 33

  - Comparing Different Time-Period Data for the Same Table .......................................................... 35
  - Comparing Measures from Different Tables ................................................................................... 37
  - Sieving the Data ............................................................................................................................... 39

**SUMMARY** ............................................................................................................................................. 40

**FURTHER READING** ................................................................................................................................ 41

**APPENDIX: DEFAULT COUNTERS RECORDED BY MDW** ........................................................................... 41

  - Performance Monitor Measurements .............................................................................................. 41

**WAIT TYPES** ........................................................................................................................................... 45

**APPENDIX: TSQL CODE EXAMPLES** ........................................................................................................ 52

  - Functions: ........................................................................................................................................ 52
    - `[snapshots].[fn_SnapshotUtcTime]` ........................................................................................... 52
    - `[snapshots].[fn_SnapshotIdPre]` ............................................................................................... 52
    - `[snapshots].[fn_SnapshotIdPost]` ............................................................................................. 52
    - `[snapshots].[fn_TimeCompare_query_stats]` .............................................................................. 53
    - `[snapshots].[fn_TableCompare_query_stats]` ........................................................................... 54
    - `[snapshots].[fn_Significant_os_wait_stats_wait_time_ms]` ...................................................... 55

  - Adding Generic T-SQL Query Collector Type TSQL .......................................................................... 56
Overview

Trouble shooting SQL Server performance problems is a common and often frustrating task for DBAs and SQL Server performance analysts (PA). Performance problem reports arrive after the fact with little supplemental information available. The typical response is either to setup limited monitoring for a few days or attempt to replicate the problem on a cloned database. The advent of the Management Data Warehouse (MDW) changes this environment by allowing continuous performance monitoring of hundreds of performance counters with little effort or performance impact. This is a significant change from reactive to proactive management. The data collected is stored in a SQL Server database that may be located on an isolated server and is easy to retrieve into products like Microsoft Excel. MDW has an API that allows recording of performance counters from components outside of SQL Server parallel to the recording of built-in SQL Server and Operating System performance counters.

Pro-active capture of data needed for performance analysis allows immediate detail analysis. The DBA or PA just needs to know when the time of the problem to bring up a treasure trove of performance counters on their monitor.

This white paper introduces you to:

- installing and running MDW
- adding custom data collections sets
- retrieving and comparing data from MDW
- sieving through the data to programmatically identify counters that may assist with the data analysis

What is a Management Data Warehouse

SQL Server 2008 introduced the Management Data Warehouse (MDW). It is a database containing a warehouse of data useful for managing SQL Server. MDW captures every standard performance counter traditionally used for performance analysis. MDW significantly lessens the work needed to monitor and trouble shoot SQL Server 2008. Many DBA’s and PA’s are unaware of MDW and the wealth of data that is now very accessible. This white paper explains the fundamentals of using MDW.

To create and use Data Management Warehouse require administrator privileges or the new data collection privileges.

---

Management Data Warehouse
MDW captures key Windows Performance Monitor (PerfMon) counters and internal SQL Server counters automatically into a SQL Server database. Running MDW continuously results in proactive capturing of data to analyze problems as reported. Prior to MDW, a DBA would typically receive a report of a problem, start running PerfMon, and hope that the problem reoccurs for analysis. With MDW, you effectively have PerfMon always running (at a low sampling rate); you just need to know when the problem occurred to do in-depth analysis.

An example of a chart produced from MDW data is shown in Figure 2.
Activating MDW

Because MDW is new in SQL Server 2008, let us walk through activating it step-by-step. SQL Server installation installs MDW automatically.

1. In SQL Server Management Studio (SSMS), open Management, Data Collection and then right-click and select Configure Management Data Warehouse.
2. If there is a Welcome screen, click past it to arrive at the [Select configuration task] dialog box. Select *Create or upgrade a management data warehouse* and then click [Next].

![Image of Configure Management Data Warehouse Wizard]

3. On the Configure Management Data Warehouse Storage screen, click [New].
4. When the [New Database] dialog box appears, enter a suitable database name. For example if you are setting up a database on a [WcAdmin] database for 2010, enter wcAdminMDW2010 as the database name and click [OK].

5. The [Configure Management Data Warehouse Storage] dialog will re-appear, click [Next].
6. The [Map Login and Users] dialog will appear. Assuming you are an administrator, there is no need to add a mapping. Click [Next].

7. At the [Complete the Wizard] dialog, click [Finish].
8. Next, you will see the [Configure Data Collection Wizard Progress] dialog. When it completes, click [Close].
9. In the SSMS Object Explorer, notice that a new database (wcAdminMDW2010) has been added. Expanding this object will show a collection of tables.
10. Now, return to Step 1: Open Management, Data Collection and then right-click and select *Configure Management Data Warehouse* menu item. This time, however, select *Set up data collection* and click [Next].
11. On the [Configure Management Data Warehouse Storage] dialog, select the new database and click [Next]. (Note: You must click the Server name ellipsis box (…) to enable the Database name combo box.)

12. At the [Complete the Wizard] dialog, click [Finish].
13. The [Configure Data Collection Wizard Progress] dialog will appear. When it is finished, click [Close].

14. Return to SSMS, and right-click to see new menu items under Management, Data Collection.
15. Under SQL Server Agent, Jobs, you will see that a new job—mdw_purge_data_[wcAdminMDW2010]—has been created. This job cleans up the database once a day; you may want to disable it if you need to retain the data or want to collect data from multiple days.

16. When you finish the wizard, data collection starts automatically.

In a production environment, you should keep the database on a separate server from the production SQL Server applications. See How to: Configure the Management Data Warehouse for Multiple Instances for further information.

**Built-in Reports**

You may have a hard time finding the built-in reports for MDW because they must be used once before they will automatically appear in the SSMS menu. To get started:

- Open SSMS and go to one of the MDW databases.
- Right-click and select the Reports menu item.
- Select Management Data Warehouse menu item.
- Select Management Data Warehouse Overview menu item.

A collection of charts and graphs showing server activity will appear; an example is shown in Figure 3.
To get more detail for any category, simply click the chart or graph you are interested in drilling down into more depth. Figure 4 shows a sample drill-down report.
The built-in reports have a default charting duration of 4 hours, which is sufficient for most troubleshooting. You also have other time intervals to choose from, as Figure 5 shows.
Advanced Troubleshooting and Analysis
MDW’s real strength is that the data is recorded and available in regular SQL Server tables. This allows you to query the data from Microsoft Excel or any other tool. For example, comparing PerfMon counters from this Monday against last Monday (or 6 months ago) becomes trivial.

Modifying MDW for Long Term Tracking
The default setup for MDW is to remove data from it when the data reaches an expiration date. This removal of data is done by calling `exec {mdw}.core.sp_purge_data`. The simplest approach is to modify the call by supplying a value to the @retention_days. A good value to use is 400 – being a year plus a month. Many databases exhibit a yearly pattern, especially if the database contains consumer sales.

The steps to do this modification are:
1. In SSMS, open SQL Server Agent / Jobs and look for job(s) starting with “mdw_purge_data…”
2. Open properties of the job that writes to the MDW database you wish to use. The database name is shown in [ ] at the end of the job’s name.
3. Click the [Steps] Page on the left side
5. Change the command to “`exec core.sp_purge_data @retention_days=400`”.

6. Click [OK]. The [Job Properties] dialog will re-appear. Click [OK] to close it. You will now retain data for 400 days.

Alternatively, in SSMS under Management / Data Collection / System Data Collection Sets you will see the three built in sets as shown in Figure 6.

**Figure 6 -- System Data Collection Sets in SSMS**

- Management
  - Policy Management
- Data Collection
  - System Data Collection Sets
    - Disk Usage
    - Query Statistics
    - Server Activity

Right clicking on these and selecting properties, see Figure 7, allow you to set the data retain period for each data collection set independently.
Extending MDW

MDW comes with three sets of data collection scripts. MDW has a simple API so you may add additional scripts to record additional information. MDW as the universal collector of performance and monitoring counters simplifies data analysis and save labor. MDW comes with hundreds of counters being collected; you do not want to recreate all of this data collection in another tool when you can simply add the missing items to MDW; items that are missing for your needs or specific to your environment or application.

A collection of built-in stored procedures\(^2\) in the [msdb] database allows modification and enhancements of MDW. To illustrate this process, we will add a set of counters to record index

fragmentation over time on all indexes using more than 100 pages that are fragmented. The TSQL shown below retrieves this information with an example of the results in Figure 8.

```sql
Select
    db_name([database_id])
    + '. '
    + Sch.[Name]
    + '. '
    + T.[Name]
    + '. '
    + Indexes.[Name] as [CounterName],
    S.[FragmentationPercentage]
FROM (Select
    [database_id],
    [object_id],
    [index_id],
    Sum(avg_fragmentation_in_percent * page_count) / Sum(page_count) as [FragmentationPercentage]
FROM sys.dm_db_index_physical_stats(
    db_id(), NULL, NULL, NULL, 'DETAILED')
Where index_type_desc IN ('CLUSTERED INDEX', 'NONCLUSTERED INDEX')
AND alloc_unit_type_desc='IN_ROW_DATA'
GROUP by [database_id],[object_id],[index_id]
HAVING Sum(page_count) > 100
and Sum(avg_fragmentation_in_percent * page_count) > 0
) S
Join sys.objects T
    On S.[object_id] = T.[object_id]
Join sys.schemas sch
    On Sch.[schema_id] = T.[schema_id]
Join Sys.Indexes
    On Indexes.[Object_id] = S.[Object_Id]
And Indexes.[Index_id] = S.[Index_Id]
```

Figure 8 -- Example of results from fragmentation index query

<table>
<thead>
<tr>
<th>CounterName</th>
<th>FragmentationPercentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>eleads.dbo.tblResponses.PK_tblResponses</td>
</tr>
<tr>
<td>2</td>
<td>eleads.dbo.tblRespondantProfile_PK_tblRespondantProfile</td>
</tr>
<tr>
<td>3</td>
<td>eleads.dbo.tblRespondantProfile__dtalink_tblResponse</td>
</tr>
<tr>
<td>4</td>
<td>eleads.dbo.tblRespondants.PK_tblRespondants</td>
</tr>
<tr>
<td>5</td>
<td>eleads.dbo.tblRespondants__dtalink_tblResponse</td>
</tr>
<tr>
<td>6</td>
<td>eleads.dbo.tblRespondants__dtalink_tblResponse</td>
</tr>
<tr>
<td>7</td>
<td>eleads.dbo.tblRespondants__dtalink_tblResponse</td>
</tr>
</tbody>
</table>

The first step is to identify the appropriate collection type by looking at the table [msdb].[dbo].[syscollector_collector_types]. There are four types supplied by the Microsoft listed below, you can create your own custom collection type.

- Generic T-SQL Query Collector Type
- Generic SQL Trace Collector Type
• Query Activity Collector Type
• Performance Counters Collector Type

The first three are SQL Server specific and we will use the first one. If you are monitoring performance of a component outside of SQL Server, create a PerfMon counter and use the last type. Examples of creating custom PerfMon counters are available in:
  • Creating COM+ PerfMon Counters to Monitor COM+ Data
  • ATL Performance Monitor Counter Wizard
  • How To: Use Custom Performance Counters from ASP.NET
  • How To: Monitor the ASP.NET Thread Pool Using Custom Counters

Creating a custom data collector
The first step is to obtain the collector type identifier through the TSQL below.

DECLARE @collector_type uniqueidentifier
SELECT @collector_type = collector_type_uid
  FROM [msdb].[dbo].[syscollector_collector_types]
  WHERE name = N'Generic T-SQL Query Collector Type'

The second step is to select how often this collection will occur. The [msdb].[dbo].[sysschedules] table lists the available schedules. The default schedules are listed below, you may create additional schedules.
  • RunAsSQLAgentServiceStartSchedule
  • CollectorSchedule_Every_5min
  • CollectorSchedule_Every_10min
  • CollectorSchedule_Every_15min
  • CollectorSchedule_Every_30min
  • CollectorSchedule_Every_60min
  • CollectorSchedule_Every_6h
  • syspolicy_purge_history_schedule
  • mdw_purge_data_schedule

For our example, we will record the information every 30 minutes shown in by the TSQL below.

DECLARE @schedule uniqueidentifier
SELECT @schedule = schedule_uid
  FROM [msdb].[dbo].[sysschedules]
  WHERE name = N'CollectorSchedule_Every_30min'

The third step is to create the collection set using the above information and the [msdb].[dbo].[sp_syscollector_create_collection_set] stored procedure. We need to retain identifiers to this collection set as shown in the TSQL below.

DECLARE @collectionsetid int
DECLARE @collectionsetuid uniqueidentifier
EXEC [msdb].[dbo].[sp_syscollector_create_collection_set]
  @name=N'Index Fragmentation Usage Report',
  @collection_mode=1, -- Non-cached mode
The last step is to supply what is to be monitored using the
[msdb].[dbo].[sp_syscollector_create_collection_item] stored procedure. Before we can do that we must define the parameters. The
[msdb].[dbo].[syscollector_collector_types] table provides a [parameter_schema] column defining what is required as shown XML 1 below.

XML 1 -- XML Schema for TSQLQueryCollector

We create the XML conforming to this schema as shown in XML 2 below. You can restrict the query to any combination of the following:

- Specific Database(s)
- All User Databases
- All System Databases
The following column names are reserved and must not be used:

- [collection_time]
- [snapshot_id]

Every column in the query should be named, the name is what will appear in the table automatically created to record the data. The table name are specified in `<OutputTable>` and created in the [Custom_Snapshot] schema. Verbose names are recommended for clarity. Consider concatenating information so that analysis and charting is easier to perform, such as shown with `[IndexName]` being a concatenation of {Database Name, Schema Name, Table Name, Index Name}. You may add multiple `<Query>` definitions into each set, this example contains only adding a single definition.

**XML 2 -- Definition of a TSQLQueryCollector**

```xml
<ns:TSQLQueryCollector xmlns:ns="DataCollectorType">
  <Query>
    <Value>
      Select
      db_name([database_id])+'.'+Sch.[Name]+'.'+T.[Name]+'.'+Indexes.[Name] as [IndexName],
      S.[FragmentationPercentage]
      FROM (Select
        [database_id], [object_id], [index_id],
        Sum(avg_fragmentation_in_percent * page_count)/Sum(page_count) as [FragmentationPercentage]
      FROM sys.dm_db_index_physical_stats( db_id(), NULL, NULL, NULL, 'DETAILED')
      WHERE index_type_desc IN ('CLUSTERED INDEX', 'NONCLUSTERED INDEX')
      GROUP BY [database_id],[object_id],[index_id]
      HAVING Sum(Page_count) &gt; 100 and Sum(avg_fragmentation_in_percent * page_count) &gt; 0 ) S
      Join Sys.objects T ON S.[object_id] = T.[object_id]
      Join Sys.schemas Sch ON Sch.[schema_id] = T.[schema_id]
    </Value>
    <OutputTable>IndexFragmentation</OutputTable>
  </Query>
</ns:TSQLQueryCollector>
```

For our final step we call the

[msdb].[dbo].[sp_syscollector_create_collection_item] stored procedure to create the data-collection-set.

---

3 Their usage will cause collection to fail and an error to appear in the log as shown in Figure 15.
DECLARE @parameters XML
Set @parameters='<ns:TSQLQueryCollector xmlns:ns="DataCollectorType">
<TSQLQueryCollector>
    <Query>
        <Value>
            Declare @collectionitem int
            Exec [msdb].[dbo].[sp_syscollector_create_collection_item]
            @name=N'Index Fragmentation',
            @parameters=@parameters,
            @collection_item_id=@collectionitem output,
            @collection_set_id=@collectionsetid,
            @collector_type_uid=@collectortype
        </Value>
    </Query>

    <OutputTable>
        IndexFragmentation
    </OutputTable>

</TSQLQueryCollector>
</ns:TSQLQueryCollector>

DElar @collectionitem int
EXEC [msdb].[dbo].[sp_syscollector_create_collection_item]
    @name=N'Index Fragmentation',
    @parameters=@parameters,
    @collection_item_id=@collectionitem output,
    @collection_set_id=@collectionsetid,
    @collector_type_uid=@collectortype

Starting and Stopping Custom Data Collection

In SSMS you will see under Management /Data Collection a new item, ‘Index Fragmentation Usage Report’ as shown in Figure 9 with a variety of actions shown on a right click.
Opening Properties, you will see the definition just entered as shown in Figure 11. You will also note that it is created in a stop state. Data-collection must be enabled prior to starting data-collection-set. Once the data-collection-set is enabled, then it will be paused when data-collection is disabled as shown in Figure 10.

**Figure 10 -- Custom Collections are paused when Data Collection is disabled**
Custom Data Collection Tables

In the example above the <OutputTable> was set to IndexFragmentation. Doing a refresh of tables in SSMS will show a new table has been created called [custom_snapshots].[IndexFragmentation] as shown in Figure 12.
An example of the data in this table is shown in Figure 13. Note that \[collection\_time\] column value for this \[snapshot\_id\] (351) is not the same for all records captured in a snapshot but records the record time for each record.

Trouble Shooting

Trouble shooting data collection sets may be done using the Log File Viewer as shown in Figure 14, where a divide by zero error occurred. The TSQL should be robust and well tested.
Including [collection_time] or [snapshot_id] columns in the query will result in the error message shown in Figure 15.

Figure 15 -- Log File Viewer error message when reserved column name used
Retrieving Data from MDW for Analysis

MDW uses a regular SQL Server database for storing information. The default installation includes hundreds of data collectors (see page 41). You may add additional custom data collector sets as needed. Figure 17 shows the key snapshots tables in the database. Custom snapshots sets join to the [snapshots_internal] table.

Always use the [snapshot_id] column to retrieving data and not [collection_time] column. The [collection_time] column will vary for the same [snapshot_id] column as seen in Figure 13 above. There is a further problem because of a bug in the distribution code, [collection_time] column is defined as DateTimeOffset and should be using GetUtcDate() but as shown in Figure 16, different time offsets may appear in the snapshot tables. Future SQL Server updates may fix this bug, so be aware of this when creating reports.

Figure 16 -- Inconsistent [Collection_Time] Offsets

<table>
<thead>
<tr>
<th>Results</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection_time</td>
<td>snapshot_id</td>
</tr>
<tr>
<td>2010-03-13 15:37:11.72000000 -08:00</td>
<td>1</td>
</tr>
<tr>
<td>2010-03-17 17:30:34.03000000 -07:00</td>
<td>1</td>
</tr>
<tr>
<td>2010-03-13 02:00:01.00000000 +00:00</td>
<td>2</td>
</tr>
<tr>
<td>2010-03-13 02:00:01.00000000 +00:00</td>
<td>3</td>
</tr>
</tbody>
</table>

The function [snapshots].[fn_SnapshotUtcTime] in the appendix provides a consistent time for each snapshot (based on UTC time recorded in the [snapshot_timetable_internal] table.
Figure 17 -- MDW Snapshots Data Model
Comparing Different Time-Period Data for the same Table

All [snapshot] and [custom_snapshot] tables use the [snapshot_id] column. This prevents a simple solution of matching snapshots by matching offsets to fail. Snapshot times may be delayed due to load, SQL Server not running, or other snapshots that may occur at the same time on occasion resulting in difficulty matching times.

Do not compare measures in the same table using [collection_time]. Collection times from the same snapshot will vary and JOINs will often fail to find a match.

One solution is to join to a table function that provides the matching relative [snapshot_id] between the two time-periods. An example of such a table function is [snapshots].[fn_TimeCompare_query_stats] in the appendix.

Figure 18 shows an example of the results from this table function and illustrate some of the problems that may be encountered including:

- No data available (the NULL shown below)
- Time fluctuations (data as collected between 3.6 and 5.7 seconds after the quarter hour)

Figure 18 -- Results of a Snapshot_Id matching function

<table>
<thead>
<tr>
<th>Results</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot_IdA</td>
<td>Snapshot_IdB</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>17</td>
<td>23</td>
</tr>
</tbody>
</table>

An example of using this function in a query is shown below with an example of the results in Figure 19. The use of the table function simplifies the construction of comparison queries.

```sql
Select R.[Minutes],
    Sum(A.Total_Logical_Reads) as TimeA,
    Sum(B.Total_Logical_Reads) as TimeB
Join SnapShots.[Query_Stats] A
On Snapshot_IdA=A.Snapshot_Id
Join SnapShots.[Query_Stats] B
```
On Snapshot_IdB=B.Snapshot_Id 
Group by Minutes 
order by Minutes 

**Figure 19 -- Results of Period Comparison of Counters in the same table**

<table>
<thead>
<tr>
<th>Minutes</th>
<th>TimeA</th>
<th>TimeB</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>735</td>
<td>79755272</td>
</tr>
<tr>
<td>47</td>
<td>750</td>
<td>89050298</td>
</tr>
<tr>
<td>48</td>
<td>765</td>
<td>91573518</td>
</tr>
<tr>
<td>49</td>
<td>780</td>
<td>95915181</td>
</tr>
<tr>
<td>50</td>
<td>795</td>
<td>102761385</td>
</tr>
<tr>
<td>51</td>
<td>810</td>
<td>106375987</td>
</tr>
<tr>
<td>52</td>
<td>825</td>
<td>108833211</td>
</tr>
<tr>
<td>53</td>
<td>840</td>
<td>111931767</td>
</tr>
<tr>
<td>54</td>
<td>855</td>
<td>114882093</td>
</tr>
<tr>
<td>55</td>
<td>870</td>
<td>117639171</td>
</tr>
<tr>
<td>56</td>
<td>885</td>
<td>117659310</td>
</tr>
<tr>
<td>57</td>
<td>900</td>
<td>124279485</td>
</tr>
<tr>
<td>58</td>
<td>915</td>
<td>127143940</td>
</tr>
<tr>
<td>59</td>
<td>930</td>
<td>125584264</td>
</tr>
<tr>
<td>60</td>
<td>945</td>
<td>89518819</td>
</tr>
<tr>
<td>61</td>
<td>960</td>
<td>105161898</td>
</tr>
<tr>
<td>62</td>
<td>975</td>
<td>111100773</td>
</tr>
<tr>
<td>63</td>
<td>990</td>
<td>473879287</td>
</tr>
</tbody>
</table>

The chart in Figure 20 compares the operating system’s Average Disk Queue Length with and without RCSI using the default MDW counters.
Comparing Measures from Different Tables

The execution of snapshots on different schedules present some challenges for comparison. Further challenges arise from one table recording one datum measure and another table recording multiple data measures for the same time-period. There are two scenarios requiring analysis:

- Comparing measures from the same time-period
- Comparing measures from the different time-periods, usually done in conjunction with the analysis above.

Comparing measures from the same time-period from different tables requires the conversion of [snapshots_id] into time-periods and not points of time. The table function [snapshots][fn_TableCompare_query_stats] in the appendix provides a robust conversion to time-periods and returns data illustrated in Figure 21. Time-periods overlapping will occur for a small number of records.
Using this table function is similar to the above but requires appropriate aggregation of data based on minutes offset from the start of each period. The TSQL below illustrates this with results show in Figure 22.

```
Select R.Minutes, R.BaseTime,
    AVG(s.num_of_Reads) as Avg_num_of_Reads,
    AVG(D.usedpages) as Avg_usedpages,
    Avg(D.pages) as Avg_pages
From [snapshots].[fn_TableCompare_disk_usage]('2010-03-18',24) R
JOIN [snapshots].[disk_usage] D ON R.Snapshot_id=D.Snapshot_id
JOIN [snapshots].[io_virtual_file_stats] S ON R.Snapshot_id=S.Snapshot_id
JOIN [snapshots].fn_SnapshotUtcTime(S.Snapshot_id) between R.StartInterval and R.EndInterval
WHERE s.LOGICAL_FILE_NAME='MSDBDATA'
AND D.Database_Name='msdb'
GROUP BY R.Minutes, R.BaseTime
```

When the tables-joins are reversed as shown in the TSQL below, different results are delivered as shown in Figure 23. Often table-join reversals make significant differences visible.

```
Select R.Minutes, R.BaseTime,
    AVG(s.num_of_Reads) as Avg_num_of_Reads,
    AVG(D.usedpages) as Avg_usedpages,
    Avg(D.pages) as Avg_pages
From [snapshots].[fn_TableCompare_io_virtual_file_stats]
    ('2010-03-18',24) R
JOIN [snapshots].[io_virtual_file_stats] S ON R.Snapshot_id=S.Snapshot_id
JOIN [snapshots].[disk_usage] D
```

<table>
<thead>
<tr>
<th>Minutes</th>
<th>BaseTime</th>
<th>Avg_num_of_Reads</th>
<th>Avg_usedpages</th>
<th>Avg_pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>2010-03-17 17:30:32.7496326 -07:00</td>
<td>187</td>
<td>1937</td>
<td>1645</td>
</tr>
<tr>
<td>60</td>
<td>2010-03-17 18:00:06.235781 -07:00</td>
<td>187</td>
<td>2175</td>
<td>1822</td>
</tr>
</tbody>
</table>
Comparing measures from the different time-periods, usually done in conjunction with the analysis above, means we have to use both [snapshots].[fn_TimeCompare_query_stats] and [snapshots].[fn_TableCompare_query_stats] as shown in the TSQL example below.

```sql
SELECT R.Minutes, R.BaseTime,
       AVG(s.num_of_Reads) AS Avg_num_of_Reads,
       AVG(D.usedpages) AS Avg_usedpages,
       AVG(D.pages) AS Avg_pages
FROM [snapshots].[fn_TableCompare_disk_usage] ('2010-03-19', 24) R
   ON R2.Snapshot_IdA = R.Snapshot_id
JOIN [snapshots].[io_virtual_file_stats] S
   ON [snapshots].[fn_SnapshotUtcTime](S.Snapshot_id)
   between R.StartInterval and R.EndInterval
WHERE s.LOGICAL_FILE_NAME = 'MSDBDATA'
AND D.Database_Name = 'msdb'
GROUP BY R.Minutes, R.BaseTime
```

The [snapshots].[fn_TableCompare_query_stats] gives us the two sets of [snapshot_id] that we are interested in. One set of the [snapshot_id] from this table function maps to the base table using [snapshot_id] and the other is used by [snapshots].[fn_TableCompare_query_stats] to obtain the start and end of the time-periods for extracting data from other tables based on time.

**Sieving the data**

In the prior section, we looked at how we can retrieve and compare data from the hundreds of counters collected. If we look at cross product of counters resulting from comparing counters to each other, we end up with more than one hundred thousand data results or charts to examine. The key to sieving data is to realize that a flat line is not interesting. Using minimum and maximum values of counters allow the elimination of many counters as not being interesting.

We will use the [snapshots].[os_wait_stats] table to illustrate the process and create a function [snapshots].[fn_Significant_os_wait_stats_wait_time_ms] that returns counter names showing more than a specific variation of value order in descending order as shown in Figure 24.
Each counter column in each table needs examination. With a little code writing doing business intelligence on these counters, the significant items can be rank-ordered and automatically charted. The benefit of sieving is faster and more thorough analysis of this treasure trove of data.

**Summary**

MDW is new to SQL Server 2008 and compare to the effort required with earlier tools, makes the life of a DBA or SQL Performance Analyst much easier. MDW puts all of the data into SQL Server Tables and has an API that allows additional performance counters to be gathered. Retrieving data is clean and simple with the use of a few adjunct functions described in this white paper and provided in the Appendix.

For both software products of ISVs and internally developed systems, the use of MDW for continuous monitoring of production systems will provide sufficient data to identify the nature of performance problems without the need of replicating conditions to obtain the counters needed for analysis. This pro-active capture of data results in faster resolution of issues with reduced hours required. When a product or system is running in a shared or virtual environment, MDW allows identification of interactions between components. For example, was the poor performance due to another application consuming available resources?

No released tool currently automates the analysis of this treasure trove of performance data. With MDW’s richness, it is just a matter of time until such tools become available.

The next generation of SQL Server 2008 based products and applications will increasing add custom data collection to their product as a mechanism to reduce support costs and improve customer satisfaction.
Further Reading

- How to Create Custom Data Collections for SQL Server Data Management Warehouse
- Data Collection
- Using SQL Server 2008 Management Data Warehouse for database monitoring in my application

Bibliography

- Accelerated SQL Server 2008 (APress, 2008), Robert E. Walters, Michael Coles, Robert Rae, Fabio Claudio Ferracchiati, Donald Farmer
- Microsoft SQL Server™ 2008 Internals: The Storage Engine (Microsoft Press, 2009), Kalen Delaney
- Professional SQL Server 2008 Administration (Wiley, 2009), Brian Knight, Katen Parel, Steven Wort, et al.
- SQL Server MVP Deep Dives (Manning, 2010), Paul Nielsen, Kalen Delaney, et al.

Appendix: Default Counters recorded by MDW

A default installation of MDW records the following counters. They are listed here because they do not appear to be documented elsewhere.

Performance Monitor Measurements

- \LogicalDisk(_Total)
  - \% Free Space
  - \% Disk Time
  - \Avg. Disk Queue Length
  - \Avg. Disk Read Queue Length
  - \Avg. Disk Write Queue Length
  - \Disk Writes/sec
  - \Avg. Disk sec/Write
  - \Avg. Disk sec/Transfer
  - \Avg. Disk sec/Read
  - \Disk Bytes/sec
  - \Disk Reads/sec
  - \Split IO/Sec
  - Then for each disk
- \Memory
  - \% Committed Bytes In Use
- Available Mbytes
- Cache Bytes
- Pages/sec
- PhysicalDisk(_Total)
  - % Disk Time
  - % Idle Time
  - Avg. Disk Bytes/Read
  - Avg. Disk Bytes/Write
  - Avg. Disk Queue Length
- Processor(_Total)
  - % Interrupt Time
  - % Processor Time
- SQLServer:Buffer Manager
  - Buffer cache hit ratio
- System
  - Processor Queue Length
- Memory
  - Available Bytes
  - Cache Faults/sec
  - Committed Bytes
  - Free & Zero Page List Bytes
  - Modified Page List Bytes
  - Page Faults/sec
  - Page Reads/sec
  - Pool Nonpaged Bytes
  - Pool Paged Bytes
  - Standby Cache Core Bytes
  - Standby Cache Normal Priority Bytes
  - Standby Cache Reserve Bytes
  - Write Copies/sec
  - Process(MsDtsSrvr)
  - % Processor Time
  - % Processor Time
- Network Interface(- card type -)
  - Bytes Total/sec
  - Output Queue Length
- Process$(TARGETPROCESS))
  - % User Time
  - Creating Process ID
  - Elapsed Time
  - Handle Count
  - ID Process
  - IO Data Bytes/sec
  - IO Data Operations/sec
  - IO Other Operations/sec
  - IO Read Bytes/sec
  - IO Read Operations/sec
- \IO Write Bytes/sec
- \IO Write Operations/sec
- \Page Faults/sec
- \Page File Bytes Peak
- \Pool Paged Bytes
- \Private Bytes
- \% Privileged Time
- \Thread Count
- \Virtual Bytes
- \Virtual Bytes Peak
- \Working Set
- \IO Other Bytes/sec
- \Page File Bytes
- \Pool Nonpaged Bytes
- \Priority Base
- \Working Set – Private
- \Working Set Peak
- And the same for each of the following:
  - \Process(_Total)
  - \Process(Idle)
  - \Process(csrss#?)
  - \Process(csrss)
  - \Process(DCEEXEC)
  - \Process(DCEEXEC#?)
  - \Process(explorer)
  - \Process(fdhost)
  - \Process(fdlauncher)
  - \Process(HbaDiscSrvr)
  - \Process(HbaHsMgr)
  - \Process(jucheck)
  - \Process(Isass)
  - \Process(lsm)
  - \Process(msdttc)
  - \Process(MsDtsSrvr)
  - \Process(RMServer)
  - \Process(services)
  - \Process(SLsvc)
  - \Process(smss)
  - \Process(spoolsv)
  - \Process(SQLAGENT)
  - \Process(sqlwriter)
  - \Process(svchost#?)
  - \Process(System)
  - \Process(taskeng)
  - \Process(taskmgr)
  - \Process(wininit)
  - \Process(winlogon#1)
  - \Process(WmiApSrv)
- \Process(WmiPrvSE#?)
- \Process(WmiPrvSE)
- \Process(cmd)

- \Server Work Queues(?)
  - \Queue Length

- \SQLServer:Buffer Manager
  - \Page life expectancy
  - Stolen pages

- \SQLServer:Databases(_Total)
  - Active Transactions
  - Transactions/sec
  - \SQLServer:Databases(Costco0313)\Active Transactions

- \SQLServer:General Statistics
  - Logins/sec
  - Logouts/sec

- \SQLServer:General Statistics
  - Processes blocked
  - Transactions
  - User Connections

- \SQLServer:Memory Manager
  - Memory Grants Outstanding
  - Memory Grants Pending

- \SQLServer:Plan Cache(Object Plans)
  - Cache Hit Ratio

- \SQLServer:Plan Cache(SQL Plans)
  - Cache Hit Ratio

- \SQLServer:Plan Cache(Temporary Tables & Table Variables)
  - Cache Hit Ratio

- \SQLServer:SQL Statistics
  - Auto-Param Attempts/sec
  - Batch Requests/sec
  - Failed Auto-Params/sec
  - SQL Attention rate
  - SQL Compilations/sec
  - SQL Re-Compilations/sec

- \SQLServer:Transactions
  - Free Space in tempdb (KB)

- \SQLServer:Workload Group Stats(default)
  - Active requests

- \SQLServer:Workload Group Stats(default)
  - Blocked tasks
  - CPU usage %

- \SQLServer:Workload Group Stats(internal)
  - Active requests
  - Blocked tasks
  - CPU usage %

- \Server Work Queues(Blocking Queue)
Queue Length

System
  File Control Bytes/sec

SQLServer:Databases(tempdb)
  Active Transactions
  Transactions/sec

SQLServer:General Statistics
  Active Temp Tables
  Logical Connections

SQLServer:Plan Cache(_Total)
  Cache Hit Ratio

System
  File Control Operations/sec
  File Read Bytes/sec
  File Read Operations/sec
  File Write Bytes/sec
  File Write Operations/sec

Wait Types

Three measurements available for each wait type:
  Count
  Wait Time in MSec
  Signal Wait Time in MSec

The wait types are:
  Backup
    BACKUP
    BACKUP_CLIENTLOCK
    BACKUP_OPERATOR
    BACKUPBUFFER
    BACKUPIO
    BACKUPTHREAD
    DISKIO_SUSPEND
  Buffer I/O
    ASYNC_DISKPOOL_LOCK
    ASYNC_IO_COMPLETION
    FCB_REPLICA_READ
    FCB_REPLICA_WRITE
    IO_COMPLETION
    PAGEIOLATCH_DT
    PAGEIOLATCH_EX
    PAGEIOLATCH_KP
    PAGEIOLATCH_NL
    PAGEIOLATCH_SH
    PAGEIOLATCH_UP
    REPLICA_WRITES
- Buffer Latch
  - PAGELatch_DT
  - PAGELatch_EX
  - PAGELatch_KP
  - PAGELatch_NL
  - PAGELatch_SH
  - PAGELatch_UP
- Compilation
  - RESOURCE_SEMAPHORE_MUTEX
  - RESOURCE_SEMAPHORE_QUERY_COMPILE
  - RESOURCE_SEMAPHORE_SMALL_QUERY
- CPU
  - CPU
  - SOS_SCHEDULER_YIELD
- Full Text Search
  - MSSEARCH
  - SOAP_READ
  - SOAP_WRITE
- Idle
  - BROKER_EVENTHANDLER
  - BROKER_RECEIVE_WAITFOR
  - BROKER_TRANSMITTER
  - CHECKPOINT_QUEUE
  - CHKPT
  - CLR_AUTO_EVENT
  - CLR_MANUAL_EVENT
  - FSAGENT
  - KSOURCE_WAKEUP
  - LAZYWRITER_SLEEP
  - LOGMGR_QUEUE
  - ONDEMAND_TASK_QUEUE
  - REQUEST_FOR_DEADLOCK_SEARCH
  - RESOURCE_QUEUE
  - SERVER_IDLE_CHECK
  - SLEEP_BPOOL_FLUSH
  - SLEEP_DBSTARTUP
  - SLEEP_DCOMSTARTUP
  - SLEEP_MSDBSTARTUP
  - SLEEP_SYSTEMTASK
  - SLEEP_TASK
  - SLEEP_TEMPDBSTARTUP
  - SNI_HTTP_ACCEPT
  - SQLTRACE_BUFFER_FLUSH
  - TRACEWRITE
  - WAIT FOR RESULTS
  - WAITFOR TASKSHUTDOWN
  - XE_DISPATCHER_WAIT
  - XE_TIMER_EVENT
- Latch
  - DEADLOCK_ENUM_MUTEX
  - INDEX_USAGE_STATS_MUTEX
  - LATCH_DT
  - LATCH_EX
  - LATCH_KP
  - LATCH_NL
  - LATCH_SH
  - LATCH_UP
  - VIEW_DEFINITION_MUTEX
- Lock
  - LCK_M_BU
  - LCK_M_IS
  - LCK_M_IU
  - LCK_M_IX
  - LCK_M_RIn_NL
  - LCK_M_RIn_S
  - LCK_M_RIn_U
  - LCK_M_RIn_X
  - LCK_M_RS_S
  - LCK_M_RS_U
  - LCK_M_RX_S
  - LCK_M_RX_U
  - LCK_M_RX_X
  - LCK_M_S
  - LCK_M_SCH_M
  - LCK_M_SCH_S
  - LCK_M_SIU
  - LCK_M_SIX
  - LCK_M_U
  - LCK_M_UIX
  - LCK_M_X
- Logging
  - LOGBUFFER
  - LOGMGR
  - LOGMGR_FLUSH
  - LOGMGR_reserve_append
  - WRITELOG
- Memory
  - CMEMTHREAD
  - LOWFAIL_MEMMGR_QUEUE
  - RESOURCE_SEMAPHORE
  - SOS_RESERVEDMEMBLOCKLIST
  - SOS_VIRTUALMEMORY_LOW
  - UTIL_PAGE_ALLOC
- Network I/O
  - ASYNC_NETWORK_IO
  - DBMIRROR_SEND
- DTC_STATE
- MSQL_DQ
- NET_WAITFOR_PACKET
- OLEDB
- Other
  - ABR
  - BAD_PAGE_PROCESS
  - BROKER_CONNECTION_RECEIVE_TASK
  - BROKER_ENDPOINT_STATE_MUTEX
  - BROKER_INIT
  - BROKER_MASTERSTART
  - BROKER_REGISTERALLENDPOINTS
  - BROKER_SHUTDOWN
  - BROKER_TASK_STOP
  - BUILTIN_HASHKEY_MUTEX
  - CHECK_PRINT_RECORD
  - CURSOR
  - CURSOR_ASYNC
  - DAC_INIT
  - DBCC_COLUMN_TRANSLATION_CACHE
  - DBMIRROR_DBM_EVENT
  - DBMIRROR_DBM_MUTEX
  - DBMIRROR_EVENTS_QUEUE
  - DBMIRROR_WORKER_QUEUE
  - DBMIRORING_CMD
  - DBTABLE
  - DEADLOCK_TASK_SEARCH
  - DEBUG
  - DISABLE_VERSIONING
  - DLL_LOADING_MUTEX
  - DROPTEMP
  - DUMP_LOG_COORDINATOR
  - DUMP_LOG_COORDINATOR_QUEUE
  - DUMPTRIGGER
  - EC
  - EE_PMOLOCK
  - EE_SPECPROC_MAP_INIT
  - ENABLE_VERSIONING
  - ERROR_REPORTING_MANAGER
  - EXECUTION_PIPE_EVENT_INTERNAL
  - FAILPOINT
  - FT_RESTART_CRAWL
  - FT_RESUME_CRAWL
  - FULLTEXT GATHERER
  - GUARDIAN
  - HTTP_ENDPOINT_COLLCREATE
  - HTTP_ENUMERATION
  - HTTP_START
- IMP_IMPORT_MUTEX
- IMPROV_IOWAIT
- INTERNAL_TESTING
- IO_AUDIT_MUTEX
- KTM_ENLISTMENT
- KTM_RECOVERY_MANAGER
- KTM_RECOVERY_RESOLUTION
- MIRROR_SEND_MESSAGE
- MISCELLANEOUS
- MSQL_SYNC_PIPE
- MSQL_XP
- PARALLEL_BACKUP_QUEUE
- PRINT.Rollback_PROGRESS
- QNMANAGER_ACQUIRE
- QPJOB_KILL
- QPJOB_WAITFOR_ABORT
- QRY_MEM_GRANT_INFO_MUTEX
- QUERY_ERRHD_SERVICE_DONE
- QUERY_EXECUTION_INDEX_SORT_EVENT_OPEN
- QUERY_NOTIFICATION_MGR_MUTEX
- QUERY_NOTIFICATION_SUBSCRIPTION_MUTEX
- QUERY_NOTIFICATION_TABLE_MGR_MUTEX
- QUERY_NOTIFICATION_UNittest_MUTEX
- QUERY_OPTIMIZER_PRINT_MUTEX
- QUERY_REMOTE_BRICKS_DONE
- QUERY_TRACEOUT
- RECOVER_CHANGEDB
- REPL_CACHE_ACCESS
- REPL_SCHEMA_ACCESS
- REQUEST_DISPENSER_PAUSE
- SEC_DROP_TEMP_KEY
- SEQUENTIAL_GUID
- SHUTDOWN
- SNI_CRITICAL_SECTION
- SNI_HTTP_WAITFOR_0_DISCON
- SNI_LISTENER_ACCESS
- SNI_TASK_COMPLETION
- SOS_CALLBACK_REMOVAL
- SOS_DISPATCHER_MUTEX
- SOS_LOCALALLOCATORLIST
- SOS_OBJECT_STORE_DESTROY_MUTEX
- SOS_PROCESS_AFFINITY_MUTEX
- SOS_STACKSTORE_INIT_MUTEX
- SOS_SYNC_TASK_ENQUEUE_EVENT
- SOSHOST_EVENT
- SOSHOST_INTERNAL
- SOSHOST_MUTEX
- SOSHOST_RWLOCK
- SOSHOST_SEMAPHORE
- SOSHOST_SLEEP
- SOSHOST_TRACELock
- SOSHOST_WAITFORDONE
- SQLSORT_NORMMUTEX
- SQLSORT_SORTMUTEX
- SQLTRACE_LOCK
- SQLTRACE_SHUTDOWN
- SQLTRACE_WAIT_ENTRIES
- SRVPROC_SHUTDOWN
- TEMPOBJ
- THREADPOOL
- TIMEPRIV_TIMEPERIOD
- VIA_ACCEPT
- WAITSTAT_MUTEX
- WCC
- WORKTBL_DROP
- XE_BUFFERMGR_ALLPROCESSED_EVENT
- XE_BUFFERMGR_FREEBUF_EVENT
- XE_DISPATCHER_JOIN
- XE_MODULEMGR_SYNC
- XE_OLS_LOCK
- XE_SERVICES_MUTEX
- XE_SESSION_CREATE_SYNC
- XE_SESSION_SYNC
- XE_STM_CREATE
- XE_TIMER_MUTEX
- XE_TIMER_TASK_DONE

- Parallelism
  - CXPACKET
  - EXCHANGE
  - EXECSYNC

- SQLCLR
  - ASSEMBLY_LOAD
  - CLR_CRST
  - CLR_JOIN
  - CLR_MEMORY_Spy
  - CLR_MONITOR
  - CLR_RWLOCK.Reader
  - CLR_RWLOCK_WRITER
  - CLR_SEMAPHORE
  - CLR_TASK_START
  - CLRHOST_STATE_ACCESS
  - FS_GARBAGE_COLLECTOR_SHUTDOWN
  - SQLCLR_APPDOMAIN
  - SQLCLR_ASSEMBLY
  - SQLCLR_DEADLOCK_DETECTION
  - SQLCLR_QUANTUM_PUNISHMENT
- Transaction
  - DTC
  - DTC_ABORT_REQUEST
  - DTC_RESOLVE
  - DTC_TMDOWN_REQUEST
  - DTC_WAITFOR_OUTCOME
  - MSQL_XACT_MGR_MUTEX
  - MSQL_XACT_MUTEX
  - TRAN_MARKLATCH_DT
  - TRAN_MARKLATCH_EX
  - TRAN_MARKLATCH_KP
  - TRAN_MARKLATCH_NL
  - TRAN_MARKLATCH_SH
  - TRAN_MARKLATCH_UP
  - TRANSACTION_MUTEX
  - XACT_OWN_TRANSACTION
  - XACT_RECLAIM_SESSION
  - XACTLOCKINFO
  - XACTWORKSPACE_MUTEX

- User Waits
  - WAITFOR
Appendix: TSQL Code Examples

Functions:
The functions below are beneficial to add to the MDW database to simplify TSQL retrieving and comparing data.

`[snapshots].[fn_SnapshotUtcTime]`
This function returns the time associated with a snapshot id. The records inserted with this [snapshot_id] may not be the same value.

```sql
CREATE FUNCTION [snapshots].[fn_SnapshotUtcTime] (
    @snapshot_Id int
) RETURNS datetimeoffset AS
BEGIN
Declare @snapshot_time datetimeoffset
select @snapshot_time = A.snapshot_time
from [core].[snapshot_timetable_internal] A
join [core].[snapshots_internal] B on A.Snapshot_time_id = B.Snapshot_time_id
where B.Snapshot_id = @Snapshot_id
RETURN @Snapshot_time
END
```

`[snapshots].[fn_SnapshotIdPre]`
This function returns the last [snapshot_id] before or at the specified time.

```sql
CREATE FUNCTION [snapshots].[fn_SnapshotIdPre] (
    @SnapShotBefore DateTimeOffset
) RETURNS int AS
BEGIN
Declare @snapshot_id int
select @snapshot_id = B.snapshot_id
from [core].[snapshot_timetable_internal] A
join [core].[snapshots_internal] B on A.Snapshot_time_id = B.Snapshot_time_id
where A.snapshot_time <= @SnapShotBefore
If @Snapshot_id is null
    Set @Snapshot_id = 1
RETURN @Snapshot_id
END
```

`[snapshots].[fn_SnapshotIdPost]`
This function returns the last [snapshot_id] after or at the specified time.
CREATE FUNCTION snapshots.fn_SnapshotIdPost
(@SnapShotAfter DateTimeOffset)
RETURNS int
AS
BEGIN
Declare @snapshot_id int
select @snapshot_id = B.snapshot_id
from [core].[snapshot_timetable_internal] A
join [core].[snapshots_internal] B
on A.Snapshot_time_id = B.Snapshot_time_id
where A.snapshot_time >= @SnapShotAfter
If @Snapshot_id Is null
Select @Snapshot_id = max(Snapshot_id) from [core].[snapshots_internal]
RETURN @Snapshot_id
END

snapshots.fn_TimeCompare_query_stats
This is an example of a function for comparing counters from different time-periods. The [Minutes] column is the time in minutes from the start of each period using the first period as a reference. This function is a template for each snapshot table or custom_snapshot table.

CREATE FUNCTION snapshots.fn_TimeCompare_query_stats
(@StartTimeUtcA DateTimeOffset, @StartTimeUtcB DateTimeOffset, @Hours int)
RETURNS @ret TABLE
(
    [Snapshot_IdA] int,
    [Snapshot_IdB] int,
    [Minutes] int,
    [BaseDateTime] DateTimeOffset
)
AS
BEGIN
Declare @FirstId int
Declare @LastId int
Set @FirstId = snapshots.fn_SnapshotIdPre(@StartTimeUtcA)
Set @LastId = DateAdd(hh, @hours, @StartTimeUtcA)
Insert into @ret (Snapshot_IdA, [Minutes], [BaseDateTime])
Select distinct Snapshot_id,
DateDiff(mi, @StartTimeUtcA, [snapshots].[fn_SnapshotUtcTime](Snapshot_id)),
    [snapshots].[fn_SnapshotUtcTime](Snapshot_id)
from Snapshots.query_stats
Where Snapshot_id between @FirstId and @LastId
Set @FirstId = snapshots.fn_SnapshotIdPre(@StartTimeUtcB)
Set @LastId=[snapshots].[fn_SnapshotIdPre](DateAdd(hh, @hours, @StartTimeUtcB))

Update @ret
Set Snapshot_IdB=Snapshot_Id
From @ret Join
(Select distinct Snapshot_Id From SnapShots.query_stats
Where Snapshot_Id between @FirstId and @LastId) S
On DateDiff(mi, @StartTimeUtcB
,[snapshots].[fn_SnapshotUtcTime](Snapshot_id))
Between minutes-7 and minutes+7
RETURN
END
GO

[snapshots].[fn_TableCompare_query_stats]
This is an example of a function for getting time-ranges from a table. Data collection may be
paused or stopped during the interval of interest resulting in the average duration of a time-
range being inappropriate. The solution shown below creates the table using the modal (most
common) duration seen in seconds. It is possible is some situations for the time-ranges to
overlap, however the number of overlaps will be a low percentage of the time-ranges. To
eliminate overlaps, the minimum duration could be used; this approach will result is less of the
interval of interest being covered. The modal approach appears to be the optimal solution given
the complexities that may occur.

This function is a template for each [snapshot] table or [custom_snapshot] table.

CREATE FUNCTION  [snapshots].[fn_TableCompare_query_stats]
(  
    @StartTimeUtcA DatetimeOffset,  
    @Hours int  
)  
RETURNS @ret TABLE  
(  
    [Snapshot_Id] int,  
    [Minutes] int,  
    [BaseTime] DdatetimeOffset,  
    [StartInterval] DdatetimeOffset,  
    [EndInterval] DdatetimeOffset  
)  
AS  
BEGIN  
    Declare @FirstId int  
    Declare @LastId int  
    Set @FirstId=[snapshots].[fn_SnapshotIdPre](@StartTimeUtcA)  
    Set @LastId=[snapshots].[fn_SnapshotIdPre](DateAdd(hh, @hours, @StartTimeUtcA))  
    Insert into @ret (Snapshot_Id, Minutes, [BaseTime])
    Select distinct Snapshot_id,
    DateDiff(mi, @StartTimeUtcA, [snapshots].[fn_SnapshotUtcTime](Snapshot_id))
    ,
    [snapshots].[fn_SnapshotUtcTime](Snapshot_id)
from SnapShots.query_stats
Where Snapshot_Id between @FirstId and @lastId
Declare @PeriodSeconds int
Select Top 1 @PeriodSeconds=PeriodSeconds/2
From
(Select DateDiff(ss,
    [snapshots].[fn_SnapshotUtcTime](startid),
    [snapshots].[fn_SnapshotUtcTime](endid)) As PeriodSeconds
From (Select distinct A.Snapshot_id as StartId,B.Snapshot_id as EndId
    from SnapShots.query_stats A
    Join SnapShots.query_stats B
    ON B.Snapshot_id = (Select Min(C.Snapshot_id)
    from SnapShots.query_stats C Where C.Snapshot_ID >
A.Snapshot_id)
) ) IntervalData
) IntervalSample
Group by PeriodSeconds
Order by Count(PeriodSeconds) Desc
Update @ret
Set StartInterval=DateAdd(ss,-@PeriodSeconds,[BaseTime]),
    EndInterval=DateAdd(ss, @PeriodSeconds,[BaseTime])
RETURN
END
GO

[snapshots].[fn_Significant_os_wait_stats_wait_time_ms]
This function is a template for each counter column in [snapshot] tables or [custom_snapshot] tables. This function uses the minimum and maximum values to obtain a significance indicator. More sophisticated heuristics should be considered; for example, using mean and standard deviation, correlation between regression lines.

CREATE FUNCTION [snapshots].[fn_Significant_os_wait_stats_wait_time_ms]
(@StartTime DatetimeOffset,
@EndTime DatetimeOffset,
@Significance float
)
RETURNS TABLE
AS
RETURN
SELECT
    [wait_type] as [wait_time_ms],
    Delta_wait_time_ms/Mid_wait_time_ms As Significance
FROM (SELECT [wait_type]
    ,cast(Max([wait_time_ms])-Min([wait_time_ms]) as float) as Delta_wait_time_ms
    ,cast((Max([wait_time_ms])+Min([wait_time_ms]))/2 as float) as Mid_wait_time_ms
FROM [MDW].[snapshots].[os_wait_stats]
Where [snapshots].[fn_SnapshotUtcTime](snapshot_id) between @StartTime and @EndTime
GROUP BY [wait_type]
HAVING (Max([wait_time_ms]) + Min([wait_time_ms])) / 2 > 0

Ranges
Where Delta_wait_time_ms/
    Case when Mid_wait_time_ms = 0 Then 1 Else Mid_wait_time_ms End
> @Significance

GO

Adding Generic T-SQL Query Collector Type TSQL
The script below installs (with rollback on errors) the Index Fragmentation Usage report

Begin Transaction
Begin Try
Declare @collection_set_id_27 int
Declare @collection_set_uid_28 uniqueidentifier
EXEC [msdb].[dbo].[sp_syscollector_create_collection_set] @name=N'Index Fragmentation Usage Report', @collection_mode=1, @description=N'Records fragmentation of indexes over 100 pages', @logging_level=1,
@days_until_expiration=400,
@schedule_name=N'CollectorSchedule_Every_30min',
@collection_set_id=collection_set_id_27 OUTPUT,
@collection_set_uid=collection_set_uid_28 OUTPUT
Select @collection_set_id_27, @collection_set_uid_28

Declare @collector_type_uid_29 uniqueidentifier
Select @collector_type_uid_29 = collector_type_uid From
[msdb].[dbo].[syscollector_collection_types] Where name = N'Generic T-SQL Query Collector Type';

Declare @collection_item_id_30 int
EXEC [msdb].[dbo].[sp_syscollector_create_collection_item] @name=N'Index Fragmentation', @parameters=N'\<ns:TSQLQueryCollector xmlns:ns="DataCollectorType">\<Query><Value>
  Select Cast(GetUtcDate() as DateTimeOffset) As [collection_time],
db_name([database_id])+''.''+Sch.[Name]+''.''+T.[Name]+''.''+Indexes.[Name] as [IndexName],
      S.[FragmentationPercentage]
FROM (Select
    [database_id], [object_id], [index_id],
    Sum(avg_fragmentation_in_percent * 
        page_count) / Sum(page_count) as [FragmentationPercentage]
    FROM sys.dm_db_index_physical_stats(  db_id(), NUL L, NULL, NULL,
       'DETAILED')
    Where index_type_desc IN ('CLUSTERED INDEX','NONCLUSTERED INDEX') AND alloc_unit_type_desc=''IN_ROW_DATA''
    Group by [database_id],[object_id],[index_id]
    HAVING Sum(page_count) gt 100 and Sum(avg_fragmentation_in_percent * 
        page_count) gt 0 ) S
  Join Sys.schemas sch On Sch.[schema_id] = T.[schema_id]
  Indexes.[Index_id] = S.[Index_id]
</Value><\OutputTable>
IndexFragmentation
Select @collection_item_id_30
Commit Transaction;
End Try
Begin Catch
Rollback Transaction;
DECLARE @ErrorMessage NVARCHAR(4000);
DECLARE @ErrorSeverity INT;
DECLARE @ErrorState INT;
DECLARE @ErrorNumber INT;
DECLARE @ErrorLine INT;
DECLARE @ErrorProcedure NVARCHAR(200);
SELECT @ErrorLine = ERROR_LINE(),
    @ErrorSeverity = ERROR_SEVERITY(),
    @ErrorState = ERROR_STATE(),
    @ErrorNumber = ERROR_NUMBER(),
    @ErrorMessage = ERROR_MESSAGE(),
    @ErrorProcedure = ISNULL(ERROR_PROCEDURE(), '-' );
RAISERROR (14684, @ErrorSeverity, 1 , @ErrorNumber, @ErrorSeverity, 
    @ErrorState, @ErrorProcedure, @ErrorLine, @ErrorMessage);

End Catch;
GO