DB2 monitoring enhancements for BLU Acceleration

Scott Walkty
Mike Springgay
Dan Behman
David Kalmuk

July 24, 2014

Overview

BLU Acceleration is a collection of technologies for analytic queries that was introduced in IBM® DB2® for Linux®, UNIX® and Windows® (LUW) Version 10.5. BLU Acceleration can provide significant benefits in many areas including performance, storage savings, and overall time to value. This article provides an overview of the monitoring capabilities that support BLU Acceleration. These capabilities provide insight into the behavior of the database server and assist with tuning and problem determination activities. Extensive example queries help you start monitoring workloads that take advantage of BLU Acceleration.

In this article, learn about the monitoring capabilities that support BLU Acceleration. These capabilities provide insight into the behavior of the database server and assist with tuning and problem determination activities. Example queries walk you through multiple monitoring situations that take advantage of BLU Acceleration. After reading this article, you will be well equipped to perform key monitoring tasks such as:

- Monitoring buffer pool performance for BLU Acceleration.
- Monitoring sort memory usage and performance.
- Monitoring the default workload manager (WLM) configuration that is enabled with BLU Acceleration.

This article assumes prior knowledge of BLU Acceleration and the various technologies that it employs. If you are unfamiliar with BLU Acceleration, start with the best practices paper Optimizing analytic workloads using DB2 10.5 with BLU Acceleration.
Monitoring buffer pool performance

DB2 10.5 introduced new buffer pool monitor elements for monitoring access to data pages for column-organized tables, as shown in Table 1.

Table 1. New buffer pool monitor elements for column-organized tables

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL_COL_L_READS</td>
<td>The number of column-organized data pages that were requested from the buffer pool.</td>
</tr>
<tr>
<td>POOL_COL_P_READS</td>
<td>The number of column-organized data pages that were read from the table space containers.</td>
</tr>
<tr>
<td>POOL_COL_LBP_PAGES_FOUND</td>
<td>The number of times that a requested column-organized data page was in the local buffer pool.</td>
</tr>
<tr>
<td>POOL_COL_WRITES</td>
<td>The number of times that a column-organized data page was physically written to disk.</td>
</tr>
<tr>
<td>POOL_ASYNC_COL_READS</td>
<td>The number of column-organized data pages that were read from the table space containers.</td>
</tr>
<tr>
<td>POOL_ASYNC_COL_READ_REQS</td>
<td>The number of read requests by the prefetcher to the operating system to read column-organized data.</td>
</tr>
<tr>
<td>POOL_ASYNC_COL_WRITES</td>
<td>The number of times that a column-organized data page was physically written to disk by either an asynchronous page cleaner or a prefetcher.</td>
</tr>
<tr>
<td>POOL_ASYNC_COL_LBP_PAGES_FOUND</td>
<td>The number of times that a column-organized data page was in the local buffer pool when a prefetcher attempted to access the page.</td>
</tr>
</tbody>
</table>

You can use these monitor elements to monitor access to column-organized data pages separately from the data pages for row-organized tables. Data page accesses for row-organized tables are reported by the POOL_DATA_* monitor elements, for example, POOL_DATA_L_READS. The separation makes it easier to understand the impact that column-organized data access has on the buffer pool when you are monitoring mixed workloads that access both row-organized and column-organized data. For example, if access to column-organized data pages is impacting the hit ratio for row-organized data pages or vice versa, you might consider using a separate buffer pool for the column-organized tables. You can easily combine the values of the POOL_DATA_* and POOL_COL_* elements by summation to generate reports that provide a unified set of buffer pool information that covers both row-organized and column-organized data.

When a query accesses a column-organized table, you might observe a small number of reads of row-organized data pages as reported by the POOL_DATA_L_READS monitor element. These reads correspond to access to table metadata that is stored outside the column-organized data pages.

The new monitor elements are reported by a number of interfaces, including the following table functions:

- MON_GET_BUFFERPOOL, which aggregates values for each buffer pool.
- MON_GET_DATABASE, which aggregates values for an entire database.
- MON_GET_TABLESPACE, which aggregates values for each table space.
You can use the new monitor elements to compute the hit ratio for column-organized data pages. Buffer pool hit ratios are some of the most fundamental metrics for the database server. They give an important overall measure of how effectively the system is using memory to avoid disk I/O. Compute the hit ratio for column-organized data pages as in Listing 1.

**Listing 1. Compute hit ratio for column-organized data pages**

(value of the POOL_COL_LBP_PAGES_FOUND monitor element - value of the POOL_ASYNC_COL_LBP_PAGES_FOUND monitor element) / (value of the POOL_COL_L_READS monitor element + value of the POOL_TEMP_COL_L_READS monitor element)

The example in Listing 2 computes the overall hit ratio for the database for row-organized data pages, column-organized data pages, and both row and column-organized data pages combined.

**Listing 2. Compute buffer pool hit ratios**

WITH POOL_DATA( ROW_DATA_PAGES_FOUND, COL_DATA_PAGES_FOUND, DATA_PAGES_FOUND, ROW_DATA_L_READS, COL_DATA_L_READS, DATA_L_READS ) AS (SELECT SUM(POOL_DATA_LBP_PAGES_FOUND - POOL_ASYNC_DATA_LBP_PAGES_FOUND) AS ROW_DATA_PAGES_FOUND, SUM(POOL_COL_LBP_PAGES_FOUND - POOL_ASYNC_COL_LBP_PAGES_FOUND) AS COL_DATA_PAGES_FOUND, SUM(POOL_DATA_LBP_PAGES_FOUND + POOL_COL_LBP_PAGES_FOUND - POOL_ASYNC_DATA_LBP_PAGES_FOUND - POOL_ASYNC_COL_LBP_PAGES_FOUND) AS DATA_PAGES_FOUND, SUM(POOL_DATA_L_READS + POOL_TEMP_DATA_L_READS) AS ROW_DATA_L_READS, SUM(POOL_COL_L_READS + POOL_TEMP_COL_L_READS) AS COL_DATA_L_READS, SUM(POOL_DATA_L_READS + POOL_COL_L_READS + POOL_TEMP_DATA_L_READS + POOL_TEMP_COL_L_READS) AS DATA_L_READS FROM TABLE(MON_GET_DATABASE(-2)) AS T) SELECT CASE WHEN DATA_L_READS > 0 THEN DEC((FLOAT(DATA_PAGES_FOUND) / FLOAT(DATA_L_READS)) * 100, 5, 2) ELSE NULL END AS DATA_PAGE_HIT_RATIO, CASE WHEN ROW_DATA_L_READS > 0 THEN DEC((FLOAT(ROW_DATA_PAGES_FOUND) / FLOAT(ROW_DATA_L_READS)) * 100, 5, 2) ELSE NULL END AS ROW_DATA_PAGE_HIT_RATIO, CASE WHEN COL_DATA_L_READS > 0 THEN DEC((FLOAT(COL_DATA_PAGES_FOUND) / FLOAT(COL_DATA_L_READS)) * 100, 5, 2) ELSE NULL END AS COL_DATA_PAGE_HIT_RATIO FROM POOL_DATA

<table>
<thead>
<tr>
<th>DATA_PAGE_HIT_RATIO</th>
<th>ROW_DATA_PAGE_HIT_RATIO</th>
<th>COL_DATA_PAGE_HIT_RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>79.72</td>
<td>77.70</td>
<td>86.74</td>
</tr>
</tbody>
</table>

1 record(s) selected.
You can obtain the data page hit ratios from the following administrative views by using the DATA_HIT_RATIO_PERCENT column for row-organized data pages and the COL_HIT_RATIO_PERCENT column for column-organized data pages:

- SYSIBMADM.MON_DB_SUMMARY
- SYSIBMADM.MON_BP_UTILIZATION
- SYSIBMADM.MON_TBSP_UTILIZATION

The various reports in the MONREPORT module now include information for I/O that is performed on column-organized tables, including computation of hit ratios for column-organized data.

Counters for column-organized data page reads are also available per table, and per statement that accesses a table, if you create a usage list for the table. To obtain these counters, use the MON_GET_TABLE and MON_GET_TABLE_USAGE_LIST table functions and the new OBJECT_COL_L_READS and OBJECT_COL_P_READS monitor elements.

### Monitoring prefetcher performance

Analytic queries that use BLU Acceleration can read and process large amounts of data. Sequential reads of pages are executed more efficiently if the DB2 prefetchers instead of the agent are used to read the pages. Several monitor elements, as shown in Table 2, can help determine prefetcher performance for queries that use BLU Acceleration.

#### Table 2. Monitor elements for prefetcher performance

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL.COL.P_READS</td>
<td>The number of column-organized data pages that were read from the table space containers (the number of physical reads). The number comprises both synchronous (agent) and asynchronous (prefetcher) reads.</td>
</tr>
<tr>
<td>POOL_ASYNC.COL_READS</td>
<td>The number of column-organized data pages that were read from the table space containers (the number of physical reads) by a prefetcher.</td>
</tr>
<tr>
<td>SKIPPED_PREFETCH_UOW.COL.P_READS</td>
<td>The number of column-organized pages that an I/O server (prefetcher) skipped because the pages were already loaded into the buffer pool by an agent in the same unit of work.</td>
</tr>
<tr>
<td>PREFETCH_WAIT_TIME</td>
<td>The time that an application spent waiting for an I/O server (prefetcher) to finish loading pages into the buffer pool. An application waits for a prefetcher only if the prefetcher started the I/O for the requested page. Otherwise, the application reads the page directly.</td>
</tr>
</tbody>
</table>

Compute the prefetch ratio as in Listing 3.

#### Listing 3. Compute prefetch

\[
100 \times \left( \frac{\text{value of the POOL_ASYNC.COL_READS monitor element}}{\text{value of the POOL.COL.P_READS monitor element}} \right)
\]

For scan-heavy workloads, the ratio should be close to 100%, which means that most physical reads are performed by prefetchers. If this value is much lower than 100%, check the SKIPPED_PREFETCH_UOW.COL.P_READS monitor element. A high value for that monitor element means
that prefetch requests were skipped by prefetchers because the agent that submitted the request loaded the pages itself. Potential causes of this problem are too few prefetchers or a slowdown in the prefetch subsystem.

Watch the value of the `PREFETCH_WAIT_TIME` monitor element over time to see whether it is increasing. An increasing value might indicate some problem in the prefetch subsystem, for example, that prefetchers are taking a long time reading pages from disk. The value of the `PREFETCH_WAIT_TIME` monitor element increases only if an agent tries to read a page that a prefetcher is reading.

The example in Listing 4 queries prefetch monitor information for column-organized data pages.

**Listing 4. Query prefetch monitor information for column-organized data pages**

```sql
SELECT
    POOL_ASYNC_COL_READS,
    POOL_COL_P_READS,
    CASE
        WHEN POOL_COL_P_READS > 0 THEN
            DEC((FLOAT(POOL_ASYNC_COL_READS) / FLOAT(POOL_COL_P_READS)) * 100, 5, 2)
        ELSE
            NULL
    END AS COL_PREFETCH_RATIO,
    SKIPPED_PREFETCH_UOW_COL_P_READS,
    PREFETCH_WAIT_TIME
FROM TABLE(MON_GET_DATABASE(-1)) AS T
POOL_ASYNC_COL_READS  POOL_COL_P_READS  COL_PREFETCH_RATIO  SKIPPED_PREFETCH_UOW_COL_P_READS  PREFETCH_WAIT_TIME
--------------------  -----------------  -------------------  --------------------------------  ------------------
0                    5                0.00                0                                0
1 record(s) selected.
```

**Monitoring sort memory**

BLU Acceleration uses new algorithms for joining and grouping data that are extremely fast and specific to operating on column-organized data. The new algorithms use hashing techniques, and the memory these algorithms require is taken from sort memory. For this reason, queries that use BLU Acceleration require much more sort memory than queries operating on traditional row-based data. As a result, monitoring and tuning sort memory usage is critical to help ensure optimal performance of your BLU Acceleration queries.

The following factors impact the performance of sort memory consumers:

- The value of the `sortheap` database configuration parameter, which limits the number of sort memory pages that a sort memory consumer can use. A sort memory consumer is any operation that uses sort memory. Examples include sorts, hash joins, and merge joins. For a complete list of sort memory consumers, see the IBM Knowledge Center.
- The value of the `sheapthres` database manager configuration parameter and the value of the `sheapthres_shr` database configuration parameter, which control the total amount of sort memory available to sort memory consumers. BLU Acceleration requires shared sort memory, so you must set the `sheapthres` parameter to 0. The `sheapthres_shr` configuration...
parameter controls shared sort memory consumption at the database level. As overall sort memory usage approaches the value of the `sheapthres_shr` parameter, memory requests are throttled, and sort consumers might receive less memory than they request. These restrictions on memory usage can result in suboptimal performance and spilling of data to temporary tables. If overall sort memory usage exceeds the value of the parameter, queries can fail.

- The number of concurrently running sort memory consumers.

Monitoring sort memory includes monitoring the concurrency and behavior of the various sort memory consumers. The main sort memory consumers in BLU Acceleration are joins and GROUP BY operations. Monitoring sort memory also includes observing elements that monitor overall sort memory usage. All monitor elements described in this section are reported by the `MON_GET_DATABASE` table function. Monitor elements reporting total sort consumers, post-threshold sort consumers, and sort consumer overflows are also reported by other interfaces, including:

- The `MON_GET_SERVICE_SUBCLASS` table function. You can use the sort monitor elements reported by this interface to understand and tune WLM configurations. For example, use a lower concurrency limit for service classes where statements have a higher number of sort memory consumers.
- The `MON_GET_PKG_CACHE_STMT` table function. You can use the sort monitor elements reported by this interface to understand the sort memory requirements per statement.

Monitoring sort usage

The monitor elements in Table 3 report overall sort memory usage for a database.

### Table 3. Sort memory usage monitor elements

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORT_SHRHEAP_ALLOCATED</td>
<td>Total amount of shared sort memory that is allocated in the database.</td>
</tr>
<tr>
<td>SORT_SHRHEAP_TOP</td>
<td>High watermark for database-wide shared sort memory.</td>
</tr>
</tbody>
</table>

Use these elements to track sort memory usage. The `SORT_SHRHEAP_TOP` monitor element provides a good indicator of the maximum amount of memory that is needed throughout query execution in the system.

The example in Listing 5 queries the current and high watermark usage for the shared sort heap.

### Listing 5. Query usage for shared sort heap

```
SELECT SORT_SHRHEAP_ALLOCATED,
       SORT_SHRHEAP_TOP
FROM TABLE(MON_GET_DATABASE(-1)) AS T

SORT_SHRHEAP_ALLOCATED SORT_SHRHEAP_TOP
---------------------- ----------------
            0           32768
1 record(s) selected.
```
Monitoring active sort consumers

You can monitor concurrent sort memory consumers by using several monitor elements, as shown in Table 4.

Table 4. Monitor elements for active sort memory consumers

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE_SORTS</td>
<td>The number of sorts in the database that currently have a sort heap allocated.</td>
</tr>
<tr>
<td>ACTIVE_HASH_JOINS</td>
<td>The total number of hash joins that are currently running and consuming memory.</td>
</tr>
<tr>
<td>ACTIVE_OlAP_FUNCS</td>
<td>The total number of OLAP functions that are currently running and consuming sort heap memory.</td>
</tr>
<tr>
<td>ACTIVE_HASH_GRPBYs</td>
<td>The number of GROUP BY operations using hashing as their grouping method that are currently running and consuming sort heap memory.</td>
</tr>
</tbody>
</table>

Monitor these elements to determine the average and maximum number of concurrent sort memory consumers. A higher concurrency should have a lower ratio of sor theap to sheapthres_shr configuration parameter values.

The example in Listing 6 computes the total number of active sort memory consumers.

Listing 6. Compute active sort memory consumers

```sql
SELECT (ACTIVE_SORTS + ACTIVE_HASH_JOINS + ACTIVE_OlAP_FUNCS + ACTIVE_HASH_GRPBYs) AS ACTIVE_SORT_CONSUMERS
FROM TABLE(MON_GET_DATABASE(-1)) AS T
```

ACTIVE_SORT_CONSUMERS

```
5
```

1 record(s) selected.

Monitoring total number of sort memory consumers

Monitor elements are provided that report the total number of sort memory consumers that executed on the database, as shown in Table 5.

Table 5. Monitor elements for total number of sort memory consumers

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL_SORTS</td>
<td>The total number of sorts that were executed.</td>
</tr>
<tr>
<td>TOTAL_HASH_JOINS</td>
<td>The total number of hash joins that were executed.</td>
</tr>
<tr>
<td>TOTAL_OlAP_FUNCS</td>
<td>The total number of OLAP functions that were executed.</td>
</tr>
<tr>
<td>TOTAL_HASH_GRPBYs</td>
<td>The total number of hashed GROUP BY operations.</td>
</tr>
<tr>
<td>TOTAL_PEDS</td>
<td>The total number of times that partial early distinct operations were executed.</td>
</tr>
</tbody>
</table>
TOTAL_PEAS | The total number of times that partial early aggregation operations were executed.

Use these elements to understand which sort memory consumers were the most active on the database server and to compute percentages for operations that were throttled or spilled data. The monitor elements in Table 5 provide context for other sort memory counters. For example, 10 sort overflows might be considered bad if only 10 sorts were performed (100% of the sorts spilled data), but if 1000 sorts were performed, only 1% of the sorts spilled data.

You can also use the total number of sorts monitor elements to approximate the number of sort consumers per statement by dividing the total counts by the number of statements that were executed.

The example in Listing 7 computes the average number of sort consumers per statement per WLM service subclass.

**Listing 7. Compute number of sort consumers**

WITH SC_SORT_CONSUMERS (SERVICE_SUPERCLASS, SERVICE_SUBCLASS, TOTAL_STMTS, TOTAL_SORT_CONSUMERS)
AS (SELECT SERVICE_SUPERCLASS_NAME, SERVICE_SUBCLASS_NAME, SUM(APP_ACT_COMPLETED_TOTAL) AS TOTAL_STMTS, SUM(TOTAL_SORTS + TOTAL_PEDS + TOTAL_PEAS + TOTAL_HASH_JOINS + TOTAL_HASH_GRPBY + TOTAL_OLAP_FUNCS) AS TOTAL_SORT_CONSUMERS
FROM TABLE(MON_GET_SERVICE_SUBCLASS(NULL, NULL, -1))AS T
GROUP BY SERVICE_SUPERCLASS_NAME, SERVICE_SUBCLASS_NAME)

SELECT SUBSTR(SERVICE_SUPERCLASS, 1, 30) SUPERCLASS, SUBSTR(SERVICE_SUBCLASS, 1, 20) SUBCLASS, TOTAL_STMTS, TOTAL_SORT_CONSUMERS, ROUND(TOTAL_SORT_CONSUMERS / NULLIF(TOTAL_STMTS, 0), 0) AS AVG_SORT_CONSUMERS_PER_STMT
FROM SC_SORT_CONSUMERS

<table>
<thead>
<tr>
<th>SUPERCLASS</th>
<th>SUBCLASS</th>
<th>TOTAL_STMTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSDFAULTMAINTENANCECLASS</td>
<td>SYSDFAULTSUBCLASS</td>
<td>960</td>
</tr>
<tr>
<td>41</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SYSDFAULTSYSTEMCLASS</td>
<td>SYSDFAULTSUBCLASS</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SYSDFAULTUSERCLASS</td>
<td>SYSDFAULTMANAGESUB</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SYSDFAULTUSERCLASS</td>
<td>SYSDFAULTSUBCLASS</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

4 record(s) selected.

**Monitoring post-threshold sort consumers**

As overall sort memory usage approaches the value of the sheapthres_shr configuration parameter, memory requests are throttled, and sort consumers might receive less memory...
than they request. This situation can lead to suboptimal performance and, in some cases, spilling data to temporary tables. Monitor elements are provided that report the number of sort memory consumers whose sort memory requests were throttled due to concurrent sort memory consumption, as shown in Table 6.

### Table 6. Monitor elements for post-threshold sort consumers

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST_SHRTHRESHOLD_SORTS</td>
<td>The total number of sorts that were throttled by the sort memory throttling algorithm.</td>
</tr>
<tr>
<td>POST_SHRTHRESHOLD_HASH_JOINS</td>
<td>The total number of hash joins that were throttled by the sort memory throttling algorithm.</td>
</tr>
<tr>
<td>POST_THRESHOLD_HASH_GRPBYS</td>
<td>The total number of hashed GROUP BY operations that were throttled by the sort memory throttling algorithm.</td>
</tr>
<tr>
<td>POST_THRESHOLD_OLAP_FUNCS</td>
<td>The total number of OLAP functions that were throttled by the sort memory throttling algorithm.</td>
</tr>
<tr>
<td>POST_THRESHOLD_PEDS</td>
<td>The total number of partial early distinct operations that were throttled by the sort memory throttling algorithm.</td>
</tr>
<tr>
<td>POST_THRESHOLD_PEAS</td>
<td>The total number of partial early aggregation operations that were throttled by the sort memory throttling algorithm.</td>
</tr>
</tbody>
</table>

Use these monitor elements with the total sort memory consumers from Table 5 to determine the percentage of sort memory operations that were throttled. If this value is high, consider taking one of the following actions:

- Increase the value of the `sheapthres_shr` parameter so that more memory is available for concurrent sort operations. Refer to the best practices paper [Optimizing analytic workloads using DB2 10.5 with BLU Acceleration](https://www.ibm.com/support/knowledgecenter/en/SSWẩu9/rdb030480.html) for guidance on configuring the `sheapthres_shr` parameter.
- Decrease the workload concurrency by adjusting WLM controls to limit the number of concurrently executing sort memory consumers.

The example in Listing 8 determines the overall percentage of sort memory consumers whose memory requests were throttled.

### Listing 8. Determine overall percentage

```sql
WITH SORT_CONSUMERS(TOTAL_SORT_CONSUMERS, TOTAL_POST_THRESHOLD_SORT_CONSUMERS) AS (SELECT (TOTAL_SORTS + TOTAL_PEDS + TOTAL_PEAS + TOTAL_HASH_JOINS + TOTAL_HASH_GRPBYS + TOTAL_OLAP_FUNCS) AS TOTAL_SORT_CONSUMERS, (POST_THRESHOLD_SORTS + POST_THRESHOLD_PEDS + POST_THRESHOLD_PEAS + POST_SHRTHRESHOLD_HASH_JOINS + POST_THRESHOLD_HASH_GRPBYS + POST_THRESHOLD_OLAP_FUNCS) AS TOTAL_POST_THRESHOLD_SORT_CONSUMERS FROM TABLE(MON_GET_DATABASE(-1)) AS T) SELECT TOTAL_SORT_CONSUMERS, TOTAL_POST_THRESHOLD_SORT_CONSUMERS, CASE WHEN TOTAL_SORT_CONSUMERS > 0 THEN DEC((FLOAT(TOTAL_POST_THRESHOLD_SORT_CONSUMERS)/FLOAT(TOTAL_SORT_CONSUMERS)) * 100, 5, 2) END AS THROTTLED_PERCENTAGE FROM SORT_CONSUMERS;
```

---

For more information, visit [ibm.com/developerWorks/](https://www.ibm.com/developerWorks/) or contact us at developerWorks®.
Monitoring sort consumer overflows (spilling)

Overflowing, or spilling, occurs when there is insufficient sort memory for a sort consumer and the consumer must spill results to a system temporary table, which might require disk access. Spilling can degrade performance and should be avoided. Several monitor elements, shown in Table 7, report the number of times that individual sort memory consumers spilled results.

### Table 7. Monitor elements for sort memory consumer overflows

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORT_OVERFLOWS</td>
<td>The total number of sorts that ran out of sort heap and might have required disk space for temporary storage.</td>
</tr>
<tr>
<td>HASH_JOIN_OVERFLOWS</td>
<td>The total number of hash joins that ran out of sort heap and might have required disk space for temporary storage.</td>
</tr>
<tr>
<td>OLAP_FUNC_OVERFLOWS</td>
<td>The total number of OLAP functions that ran out of sort heap and might have required disk space for temporary storage.</td>
</tr>
<tr>
<td>HASH_GRPBY_OVERFLOWS</td>
<td>The total number of hashed GROUP BY operations that ran out of sort heap and might have required disk space for temporary storage.</td>
</tr>
</tbody>
</table>

Use these elements with the total sort memory consumer elements from Table 5 to understand the percentage of sort memory operations that spilled results. If the percentage is high, check the post-threshold elements from Table 6. If there is a large percentage of throttled sort memory consumers, sorts might have spilled results because they did not receive the sort memory that they requested. In this case, consider taking one of the following actions:

- Increase the value of the `sheapthres_shr` parameter so that more memory is available for concurrent sort operations. Refer to the best practices paper [Optimizing analytic workloads using DB2 10.5 with BLU Acceleration](https://www.ibm.com/support/knowledgecenter/SSSHTN_10.5.0/com.ibm.db2.luw manuals.doc/dbc03000e.html) for guidance on configuring the `sheapthres_shr` parameter.
- Adjust the WLM concurrency limits for the workload to reduce the number of concurrently executing sort consumers.

If sort consumers are not being throttled, consider increasing the value of the `sortheap` parameter (the value is likely too low for the queries being run). When increasing the value of the `sortheap` parameter, take into account the concurrency of sort memory consumers. The aggregate sort memory usage across concurrently executing sort memory consumers will increase and might approach the shared sort heap threshold. Use the monitor elements described in Monitoring sort usage to see how an increase in the value of the `sortheap` parameter impacts overall sort memory usage.
consumption. You might need to increase the value of the sheaptres_shr parameter or decrease the allowable query concurrency by adjusting the WLM concurrency limits.

The example in Listing 9 determines the overall percentage of sort memory consumers that spilled results.

### Listing 9. Sort memory consumers that spilled results

```sql
WITH SORT_CONSUMERS(TOTAL_SORT_CONSUMERS, TOTAL_SORT_CONSUMER_OVERFLOWS)
AS (SELECT (TOTAL_SORTS + TOTAL_HASH_JOINS + TOTAL_HASH_GRPBYYS + TOTAL_OLP_FUNC)
AS TOTAL_SORT_CONSUMERS,
(SORT_OVERFLOWS + HASH_JOIN_OVERFLOWS + HASH_GRPBY_OVERFLOWS + OLAP_FUNC_OVERFLOWS)
AS TOTAL_SORT_CONSUMER_OVERFLOWS
FROM TABLE(MON_GET_DATABASE(-1)) AS T)
SELECT TOTAL_SORT_CONSUMERS, TOTAL_SORT_CONSUMER_OVERFLOWS,
CASE WHEN TOTAL_SORT_CONSUMERS > 0 THEN
DEC(FLOAT(TOTAL_SORT_CONSUMER_OVERFLOWS)/
FLOAT(TOTAL_SORT_CONSUMERS)) * 100, 5, 2)
ELSE NULL
END AS PERCENT_SORT_CONSUMER_OVERFLOWS
FROM SORT_CONSUMERS
TOTAL_SORT_CONSUMERS TOTAL_SORT_CONSUMER_OVERFLOWS
PERCENT_SORT_CONSUMER_OVERFLOWS
----------------------------- -----------------------------
194                             2
1.03                             0
1 record(s) selected.
```

### Monitoring table size

Column-organized data is stored in a separate table data object from that of row-organized data. You can query the size of the column-organized table data object by using the ADMIN_GET_TAB_INFO table function and approximate the size of the column-organized table data object by using the MON_GET_TABLE table function. Table 8 provides more detail.

### Table 8. Monitor elements for the size of column-organized tables

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL_OBJECT_P_SIZE</td>
<td>The amount of disk space that is physically allocated for the column-organized data in the table, reported in kilobytes. This monitor element is reported by the ADMIN_GET_TAB_INFO table function.</td>
</tr>
<tr>
<td>COL_OBJECT_L_SIZE</td>
<td>The amount of disk space that is logically allocated for the column-organized data in the table, reported in kilobytes. This monitor element is reported by the ADMIN_GET_TAB_INFO table function.</td>
</tr>
<tr>
<td>COL_OBJECT_L_PAGES</td>
<td>The number of logical pages that are used on disk by column-organized data in the table. This monitor element is reported by the MON_GET_TABLE table function.</td>
</tr>
</tbody>
</table>

Generally, the physical reported by COL_OBJECT_P_SIZE and the logical size reported by COL_OBJECT_L_SIZE are the same. One exception is if you truncate a table by using the
TRUNCATE statement with the REUSE_STORAGE parameter. In this case, storage continues to be allocated to the table following truncation, but logically, the table is considered to be empty.

The size of the table object that stores the data for row-organized tables is reported in the DATA_OBJECT_P_SIZE and DATA_OBJECT_L_SIZE monitor elements reported by the ADMIN_GET_TAB_INFO table function. For column-organized tables, this data object contains a very small amount of metadata; the column data is stored in the separate column-organized data object.

To determine the size of a column-organized table, sum the DATA and COL object sizes that are reported by the ADMIN_GET_TAB_INFO table function and ADMINTABINFO administrative view.

The example in Listing 10 queries the size of the TEST.TAB1 column-organized table.

Listing 10. Querying size

```sql
SELECT DATA_OBJECT_P_SIZE + COL_OBJECT_P_SIZE AS DATA_P_SIZE,
       DATA_OBJECT_L_SIZE + COL_OBJECT_L_SIZE AS DATA_L_SIZE
FROM TABLE(ADMIN_GET_TAB_INFO('TEST','TAB1')) AS T;
```

<table>
<thead>
<tr>
<th>DATA_P_SIZE</th>
<th>DATA_L_SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10496</td>
<td>10496</td>
</tr>
</tbody>
</table>

1 record(s) selected.

This query works equally well for row-organized tables because the values of the COL_OBJECT_P_SIZE and COL_OBJECT_L_SIZE monitor elements are 0 for row-organized tables.

The ADMIN_GET_TAB_INFO table function reports accurate size information. It can run for a long time while determining the size information, which can make it an expensive option if you wish to continuously monitor table size. You can use an alternative approach to determine the size, however. The COL_OBJECT_L_PAGES monitor element in Table 8 reports an approximate value of the number of pages that were used to store column-organized data in a table. Using this value, you can compute the approximate size of the table by multiplying by the table page size and then dividing by 1024 to convert the size to kilobytes. In most cases, the approximate size is close to the accurate size, and the approximation will perform better than using the ADMIN_GET_TAB_INFO table function. The value might be too low or too high in some cases if you performed a REORG TABLE command with the RECLAIM EXTENTS parameter or if you issued a TRUNCATE TABLE statement with the REUSE STORAGE parameter. The IBM Knowledge Center has more detail on the COL_OBJECT_L_PAGES monitor element.

The example in Listing 11 approximates the size of the column-organized TEST.TAB1 table.
Listing 11. Approximating size

```sql
SELECT ((DATA_OBJECT_L_PAGES + COALESCE(COL_OBJECT_L_PAGES, 0)) * PAGESIZE) / 1024 AS TABLE_SIZE_KB
FROM TABLE(MON_GET_TABLE('TEST', 'TAB1', -1)) AS T,
SYSCAT.TABLESPACES AS Q
WHERE T.TBSP_ID = Q.TBSPACEID
```

<table>
<thead>
<tr>
<th>TABLE_SIZE_KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1740</td>
</tr>
</tbody>
</table>

1 record(s) selected.

Monitoring table compression

Performance of queries against column-organized tables is sensitive to the tables having proper compression. You can monitor compression for a column-organized table by examining the PCTPAGESSAVED column in SYSCAT.TABLES for the table. The PCTPAGESSAVED column reports the approximate percentage of pages that were saved because of compression as compared to an estimate of the number of pages that would be required to store the table in an uncompressed row-organized structure. You can use the PCTPAGESSAVED column to compute the compression ratio for the table, as in Listing 12.

Listing 12. Compute compression ratio

\[
\text{Compression ratio} = \frac{(\text{Uncompressed size})}{(\text{Compressed size})} = \frac{1}{1 - \left(\frac{\text{value of the PCTPAGESSAVED column}}{100}\right)}
\]

The following example computes the compression ratio for the table TEST.TAB1.

First, select the PCTPAGESSAVED column from SYSCAT.TABLES, as in Listing 13.

Listing 13. Select PCTPAGESSAVED

```sql
SELECT PCTPAGESSAVED FROM SYSCAT.TABLES WHERE TABSCHEMA = 'TEST' AND TABNAME = 'TAB1'
```

<table>
<thead>
<tr>
<th>PCTPAGESSAVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
</tr>
</tbody>
</table>

1 record(s) selected.

Then, compute the compression ratio, as in Listing 14.

Listing 14. Compute compression ratio

\[
\text{Compression ratio} = \frac{1}{1 - \left(\frac{75}{100}\right)} = 4
\]

If the compression ratio is low, check the PCTENCODED column in SYSCAT.COLUMNS. The PCTENCODED column contains the percentage of values that are encoded as a result of
compression for a column in a column-organized table. This statistic measures the number of values that are compressed, not the compression ratio.

The example in Listing 15 queries the PCTENCODED value for the columns of table TEST.TAB1.

**Listing 15. Query PCTENCODED value**

```sql
SELECT SUBSTR(COLNAME, 1, 20) AS COLNAME, PCTENCODED FROM SYSCAT.COLUMNS WHERE TABSCHEMA = 'TEST' AND TABNAME = 'TAB1'
```

<table>
<thead>
<tr>
<th>COLNAME</th>
<th>PCTENCODED</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL1</td>
<td>90</td>
</tr>
<tr>
<td>COL2</td>
<td>75</td>
</tr>
<tr>
<td>COL3</td>
<td>100</td>
</tr>
</tbody>
</table>

3 record(s) selected.

The values of both the PCTPAGESSAVED and PCTENCODED columns are -1 if you did not collect statistics on the table.

**Monitoring query execution with section actuals**

Section actuals are runtime statistics that are captured during the execution of the section for an SQL statement. In particular, section actuals report the actual cardinality (number of rows) that are output from runtime operators in the access plan. You can compare this actual cardinality to the cardinality that the operator estimated. Differences between estimated and actual cardinality might indicate that the optimizer did not have sufficient information when building the plan for the statement. For example, table statistics might have been out of date.

To access section actuals for a statement:

1. Enable collection of section actuals by using the `section_actuals` configuration parameter or WLM controls.
2. Capture the execution of the statement by using an activity event monitor.
3. Explain the statement by using the `EXPLAIN_FROM_ACTIVITY` stored procedure.

For details and examples of the process for performing an explain with actuals, see the IBM Knowledge Center.

For queries against column-organized tables, section actuals are available only for operators above and including the CTQ operator. The CTQ operator in an access plan represents the transfer of data from column-organized processing to row-organized processing.

For example, consider an explain with actuals for a query that joins the column-organized table TEST.COLTAB with the row-organized table TEST.ROWTAB, as in Listing 16.

**Listing 16. Output of explain with actuals**

```sql
select * from
```
```sql
  test.coltab as a,
  test.rowtab as b
where
  a.one = b.one
```

### Access Plan:

<table>
<thead>
<tr>
<th>Access Plan</th>
<th>Total Cost</th>
<th>Query Degree</th>
<th>Rows</th>
<th>Rows Actual</th>
<th>RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>388.438</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cost</td>
<td>388.438</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>I/O</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LTQ</td>
<td>388.438</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NLJOIN</td>
<td>381.854</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CTQ</td>
<td>161.186</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TBSCAN</td>
<td>9.1376</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TABLE: TEST</td>
<td>128.422</td>
<td>Q1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Actual cardinality values are provided for the CTQ operator, for operators that appear higher in the plan (for example, NLJOIN), and for operators that access the row-organized table (for example, the TBSCAN operator on TEST.ROWTAB). Actual cardinality values are not available for operators below the CTQ operator (for example, the TBSCAN operator on the TEST.COLTAB table).

Use the actual cardinality counter reported for the CTQ operator (in this example, 3072) to understand the amount of data that is being transferred from column-organized processing to row-organized processing. In good execution plans for column-organized tables, most of the operators and the processing occur below the CTQ operator, and only a few rows flow through the CTQ operator.
Monitoring query execution with time-spent metrics

DB2 Version 9.7 introduced monitor elements that provide a hierarchical breakdown of where time is spent in the DB2 engine. DB2 10.5 introduced two new time-spent monitor elements, described in Table 9, to report time that is spent processing column-organized data.

Table 9. Time-spent monitor elements for processing column-organized data

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL_COL_TIME</td>
<td>The total elapsed time that was spent processing column-organized data.</td>
</tr>
<tr>
<td>TOTAL_COL_PROC_TIME</td>
<td>The total non-wait time that was spent processing column-organized data.</td>
</tr>
</tbody>
</table>

You can view these elements as reporting the elapsed and non-wait time that is spent in all operators below the CTQ operators in an access plan.

The column-organized time-spent monitor elements are reported by all the monitor interfaces that report time-spent monitor elements, including the following table functions:

- MON_GET_PKG_CACHE_STMT, which reports the aggregate values of time-spent elements per statement.
- MON_GET_DATABASE, which reports the aggregate values of time-spent elements for a database.

The IBM Knowledge Center has a complete list of the many monitor interfaces that report time-spent monitor elements.

You can use the TOTAL_COL_TIME monitor element with the TOTAL_SECTION_TIME monitor element to determine the percentage of query execution time that was spent in columnar processing. A large ratio of the TOTAL_COL_TIME monitor element value to the TOTAL_SECTION_TIME monitor element value suggests an efficient plan for BLU Acceleration, where most of the time is spent processing column-organized data.

The example in Listing 17 determines the ratio of column-organized data processing time to total section execution time for the query `SELECT * FROM TEST.COLTAB, TEST.ROWTAB WHERE A.ONE = B.ONE` in the package cache.
Listing 17. Determine ratio of column-organized data processing time to total section execution time

```sql
SELECT TOTAL_SECTION_TIME, TOTAL_COL_TIME,
    CASE WHEN TOTAL_SECTION_TIME > 0 THEN
        DEC((FLOAT(TOTAL_COL_TIME)/
            FLOAT(TOTAL_SECTION_TIME)))*100,5,2)
    ELSE
        NULL
    END AS PCT_COL_TIME
FROM TABLE(MON_GET_PKG_CACHE_STMT(NULL,NULL,NULL,-1)) AS T
WHERE STMT_TEXT = 'SELECT * FROM TEST.COLTAB A, TEST.ROWTAB B
    WHERE A.ONE = B.ONE'
```

<table>
<thead>
<tr>
<th>TOTAL_SECTION_TIME</th>
<th>TOTAL_COL_TIME</th>
<th>PCT_COL_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>80.00</td>
</tr>
</tbody>
</table>

1 record(s) selected.

Monitoring default WLM concurrency controls

By design, queries against column-organized tables might use a large amount of CPU and memory resources. These queries are highly parallelized and are optimized for speed. To ensure that queries against column-organized data do not overload the system when many queries are submitted simultaneously, there is a limit on the number of expensive queries that are allowed to run concurrently. The query expense is determined by the estimated cost of the query. Any query that is submitted after the concurrency limit is exceeded is queued for execution until one of the currently executing queries is complete.

Classification of expensive queries is performed by the new SYSMANAGEDQUERIES work class, which examines the estimated cost of a query and determines whether or not a query will be managed. Managed queries are mapped to the new default service subclass SYSDEFAULTMANAGEDSUBCLASS, and the new default WLM concurrency threshold SYSDEFAULTCONCURRENT is applied to queries in that service subclass. This threshold is automatically enabled for any database that you create while the DB2_WORKLOAD registry variable is set to ANALYTICS. You can manually enable the threshold for pre-existing databases. The SYSMANAGEDQUERIES work class, SYSDEFAULTMANAGEDSUBCLASS service subclass, and SYSDEFAULTCONCURRENT threshold are all new in DB2 Version 10.5.

The following sections focus on monitoring the WLM concurrency configuration. A full description of WLM is outside the scope of this article. For more details about WLM and the new default concurrency threshold, see the IBM Knowledge Center.

Monitoring default WLM concurrency configuration

WLM object definitions are stored in the system catalogs. To view the default WLM concurrency configuration, select information about the SYSMANAGEDQUERIES work class and the SYSDEFAULTCONCURRENT threshold from the SYSCAT.WORKCLASSATTRIBUTES and SYSCAT.THRESHOLDS catalog tables.

The example in Listing 18 queries the default estimated cost setting that is used to determine whether to apply the concurrency limit to queries.
Listing 18. Query the default estimated cost setting

```sql
SELECT VALUE1 AS EXPENSIVE_QUERY_COST
FROM SYSCAT.WORKCLASSATTRIBUTES
WHERE WORKCLASSNAME = 'SYSMANAGEDQUERIES' AND TYPE = 'TIMERONCOST'

EXPENSIVE_QUERY_COST
------------------------
+1.50000000000000E+005

1 record(s) selected.
```

Any query with an estimated cost that is greater than or equal to 150,000 timerons is subject to the concurrency limit.

You can view the estimated cost for any query that was executed by using the `QUERY_COST_ESTIMATE` monitor element, which is reported by the `MON_GET_PKG_CACHE_STMT` table function.

The example in Listing 19 checks whether the default concurrency threshold is enabled and queries the concurrency limit.

Listing 19. Check default concurrency threshold

```sql
SELECT MAXVALUE, ENABLED
FROM SYSCAT.THRESHOLDS
WHERE THRESHOLDNAME = 'SYSDEFAULTCONCURRENT'

MAXVALUE             ENABLED
-------------------- -------
11 Y

1 record(s) selected.
```

The output shows that the default concurrency threshold is enabled and that at most 11 queries with estimated costs that are greater than or equal to 150,000 (the value in the prior example) are allowed to run concurrently. When there are more than 11 queries with estimated costs that are greater than or equal to 150,000 timerons, the additional queries are queued for execution.

Monitoring default WLM concurrency control behavior

The default WLM concurrency control threshold queues some statements to help reduce resource contention and improve overall performance. Monitoring can help you understand the total number of statements that are impacted by the default concurrency controls and pinpoint which statements are currently executing and which are queued. As shown in Table 10, you can use the `WLM_QUEUE_TIME` and `WLM_QUEUE_ASSIGNMENTS_TOTAL` monitor elements to see the impact of queuing on your database server.

Table 10. Monitor elements for measuring WLM queue time

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>WLM_QUEUE_ASSIGNMENTS_TOTAL</td>
<td>The number of statements that have been queued.</td>
</tr>
</tbody>
</table>
These monitor elements are reported by the MON_GET_DATABASE table function and in several other monitor interfaces.

The example in Listing 20 determines the amount of time that statements have been queued on the database server, the number of statements that have been queued, and the percentage of statements that were queued relative to the total number of statements that were executed.

**Listing 20. Measuring the impact of queuing**

```sql
SELECT WLM_QUEUE_TIME_TOTAL, WLM_QUEUE_ASSIGNMENTS_TOTAL, ACT_COMPLETED_TOTAL AS TOTAL_STMTS,
CASE WHEN ACT_COMPLETED_TOTAL > 0 THEN DEC((FLOAT(WLM_QUEUE_ASSIGNMENTS_TOTAL)/FLOAT(ACT_COMPLETED_TOTAL))*100,5,2)
ELSE NULL
END AS PCT_STMTS_QUEUED
FROM TABLE(MON_GET_DATABASE(-1)) AS T
```

<table>
<thead>
<tr>
<th>WLM_QUEUE_TIME_TOTAL</th>
<th>WLM_QUEUE_ASSIGNMENTS_TOTAL</th>
<th>TOTAL_STMTS</th>
<th>PCT_STMTS_QUEUED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2229</td>
<td>0.00</td>
</tr>
</tbody>
</table>

1 record(s) selected.

You can use the new MON_GET_ACTIVITY and MON_GET_AGENT table functions to monitor which statements are currently executing and which statements are queued and their queue positions.

The example in Listing 21 lists all statements that are currently executing in the SYSDEFAULTMANAGEDSUBCLASS service class.

**Listing 21. Currently executing statements in the SYSDEFAULTMANAGEDSUBCLASS service class**

```sql
SELECT SUBSTR(STMT_TEXT, 1, 100) AS CONCURRENT_EXEC_STMTS
FROM TABLE(MON_GET_ACTIVITY(NULL,-1)) AS A, SYSCAT.SERVICECLASSES AS B
WHERE A.ACTIVITY_STATE NOT IN ('QUEUED') AND A.SERVICE_CLASS_ID = B.SERVICECLASSID AND B.SERVICECLASSNAME = 'SYSDEFAULTMANAGEDSUBCLASS'
```

select * from syscat.tables

1 record(s) selected.
These statements are blocking any statements that are queued by the SYSDEFAULTCONCURRENT threshold. The output shows that a single statement, `SELECT * FROM syscat.tables`, is executing in the SYSDEFAULTMANAGEDSUBCLASS subclass.

The example in Listing 22 lists all statements currently queued by the SYSDEFAULTCONCURRENT threshold and their positions in the queue.

**Listing 22. Statements currently queued by the SYSDEFAULTCONCURRENT threshold**

```sql
SELECT AGENT_STATE_LAST_UPDATE_TIME AS QUEUE_ENTRY_TIME,
       SUBSTR(STMT_TEXT, 1, 100) AS QUEUED_STMTS
FROM TABLE(MON_GET_ACTIVITY(NULL,-1)) AS A,
       TABLE(MON_GET_AGENT(NULL,NULL,NULL,-1)) AS B
WHERE A.APPLICATION_HANDLE = B.APPLICATION_HANDLE AND
      A.UOW_ID = B.UOW_ID AND
      A.ACTIVITY_ID = B.ACTIVITY_ID AND
      B.EVENT_OBJECT = 'WLM_QUEUE' AND
      B.EVENT_OBJECT_NAME = 'SYSDEFAULTCONCURRENT'
ORDER BY QUEUE_ENTRY_TIME ASC
```

<table>
<thead>
<tr>
<th>QUEUE_ENTRY_TIME</th>
<th>QUEUED_STMTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-08-02-14.59.20.425860</td>
<td>select count(*) from syscat.indexes</td>
</tr>
<tr>
<td>2013-08-02-15.10.30.228681</td>
<td>select * from swalkty.t7</td>
</tr>
</tbody>
</table>

2 record(s) selected.

The output shows that two statements are queued for execution by the SYSDEFAULTCONCURRENT threshold. The `SELECT count(*)` statement is queued ahead of the `SELECT *` statement.

**Monitoring referenced columns**

When considering whether queries against a row-organized table might perform better if the table were column-organized, one interesting piece of information that you can use is the number of columns that are typically referenced by queries. A wide table that has only a handful of columns referenced on average might be a good candidate for column organization because only a subset of the data in each row must be retrieved. In DB2 10.5, two monitor elements were introduced for the MON_GET_TABLE table function to provide information about the number of columns that were referenced on average when a table was accessed in a query. Table 11 describes the new time-spent monitor elements.

**Table 11. Time-spent monitor elements for processing column-organized data**

<table>
<thead>
<tr>
<th>Monitor element</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_COLUMNS_REFERENCED</td>
<td>The number of columns that were referenced during the execution of a section for an SQL statement.</td>
</tr>
<tr>
<td>SECTION_EXEC_WITH_COL_REFERENCES</td>
<td>The number of section executions that referenced columns in a table by using a scan.</td>
</tr>
</tbody>
</table>

Each time a query accesses a table, the value of the `NUM_COLUMNS_REFERENCED` monitor element is incremented by the number of columns that the query referenced in the table, and the value of
the `SECTION_EXEC_WITH_COL_REFERENCES` monitor element is increased by 1. You can use these monitor elements to compute the average number of columns that were referenced by queries that accessed the table.

Suppose that the four queries in Listing 23 were run against the TEST.ROWTAB table.

**Listing 23. Example queries**

<table>
<thead>
<tr>
<th>Query</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT COL1 FROM TEST.ROWTAB (1 column referenced)</td>
<td></td>
</tr>
<tr>
<td>SELECT COL2 FROM TEST.ROWTAB (1 column referenced)</td>
<td></td>
</tr>
<tr>
<td>SELECT COL3 FROM TEST.ROWTAB WHERE COL1 = 1 (2 columns referenced)</td>
<td></td>
</tr>
<tr>
<td>SELECT COL4 FROM TEST.ROWTAB (1 column referenced)</td>
<td></td>
</tr>
</tbody>
</table>

Compute the average number of columns that were referenced by SQL statements that accessed the TEST.ROWTAB table, as in Listing 24.

**Listing 24. Compute average number of columns referenced by SQL statements that accessed TEST.ROWTAB table**

<table>
<thead>
<tr>
<th>NUM_QUERIES_THAT_ACCESSED_TABLE</th>
<th>AVG_COLS_REFERENCED_PER_QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

The `NUM_COLUMNS_REFERENCED` and `SECTION_EXEC_WITH_COL_REFERENCES` monitor elements work as well for column-organized tables as they do for row-organized tables. For example, you can use these monitor elements to help determine the average number of columns that were referenced for each query against a column-organized table.

**Snapshot monitor interfaces**

New monitor elements for monitoring workloads in a BLU Accelerated environment are not available for the snapshot commands, snapshot APIs, and SNAP* table functions. The following new monitor elements are available only through the MON_* table function interfaces and various event monitors:

- New buffer pool monitor elements (`POOL_COL_*` monitor elements).
- New sort-related monitor elements (`*HASH_GRPBY*` monitor elements).
- Time-spent monitor elements, including the new `TOTAL_COL_TIME` monitor element and the `WLM_QUEUE_TIME_TOTAL` monitor element.
- New monitor elements for tracking referenced columns (the `NUM_COLUMNS_REFERENCED` and `SECTION_EXEC_WITH_COL_REFERENCES` monitor elements).

Use the new MON_* interfaces when monitoring workloads in a BLU Accelerated environment. Commonly used MON_* table functions are:
- MON_GET_DATABASE. Reports monitor data for an entire database.
- MON_GET_CONNECTION. Reports monitor data per connection.
- MON_GET_UOW. Reports monitor data per transaction.
- MON_GET_SERVICE_SUBCLASS. Reports monitor data per service class.
- MON_GET_WORKLOAD. Reports monitor data per workload object.
- MON_GET_ACTIVITY. Reports monitor data for all currently executing statements.
- MON_GET_PKG_CACHE_STMT. Reports monitor data for all statements in the package cache, aggregated across statement executions.
- MON_GET_AGENT. Reports information for all agents currently executing in the system.
- MON_GET_BUFFERPOOL. Reports buffer pool information.
- MON_GET_TABLESPACE. Reports table space information.
- MON_GET_CONTAINER. Reports container information.
- MON_GET_TABLE. Reports information about accessed tables.
- MON_GET_INDEX. Reports information about indexes that were used.

The IBM Knowledge Center has a complete list of monitor interfaces and elements that are reported by each interface.

Summary

This article is a starting point for monitoring workloads that take advantage of BLU Acceleration. Detailed example queries showed you how to perform key monitoring tasks involving buffer pool performance, sort memory usage, and WLM configuration.

Acknowledgements

The authors would like to acknowledge Jim Seeger, Serge Boivin, and Leslie McDonald for their review and comments on this article.
Related topics

- "Optimizing analytic workloads using DB2 10.5 with BLU Acceleration" (Best Practices, 2013) provides an overview, recommendations on hardware and software selection, guidelines for identifying the optimal workloads for BLU Acceleration, and information about capacity planning, memory, and I/O.
- The IBM Knowledge Center is the home for IBM product documentation.

© Copyright IBM Corporation 2014
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)