# The Effect of Max Active on DB2 for i Query Performance

Anil K. Madan September 19, 2011

This article explains how a high Max Active setting for a memory pool can affect the fair share of memory, thereby influencing the DB2 for i query optimizer to sometimes choose an inefficient access plan.

## Overview

You already know that a constrained memory pool on IBM i can negatively impact the performance of DB2 for i queries and SQL requests. But, I bet many of you didn't realize that the Max Active setting for a memory pool can also have a large influence on the query optimizer and as a result the performance of queries. Recently, I ran into one such situation, where the Max Active value for the query pool had unknowingly become too high, which caused some of the queries to take much longer than normal. For example, one of the SQL statements was now running for hours as compared to its normal 10 to 15 minute run. Before I go into the specifics of this query, let's more closely examine the Max Active setting and how it can influence the DB2 for i query optimizer.

### What is Max Active?

Max Active (also known as activity level) is a memory tuning parameter, which controls the maximum number of threads in a memory pool that can use the processor concurrently. Every shared and private memory pool has a Max Active value associated with it. If this value is too low, the threads may transition to the ineligible condition. If this value is too high, excessive page faulting may occur. Max Active can be viewed and changed with the WRKSYSSTS (Work with System Status) command or by using System i Navigator – Work Management – Memory Pool. The Max Setting value can also be automatically adjusted by the IBM i operating system if the performance adjuster on the IBM i partition is turned ON (system value QPFRADJ 2 or 3).

### How can Max Active influence DB2 for i query optimizer?

The execution of every query is based on the instructions stored in an access plan. The access plan is created by the DB2 for i query optimizer, prior to the query's run. This plan is based on the intersection of various factors such as database design, indexes, number of table rows, column statistics, query attributes, etc. to name a few. One important environment factor that the optimizer takes into account when creating an access plan is the share of memory available for that query (its fair share of the memory). This fair share of memory calculation keeps the optimizer from over committing memory for a given query, and it allows the optimizer to accordingly consider more or

less memory intensive methods. The fair share of memory value is calculated differently depending on whether the query is being processed by the Classic Query Engine (CQE) or the SQL Query Engine (SQE). The following algorithms are used to calculate the fair share value:

### i 7.1 enhancement

In i 7.1, a new PTF - MF54009 is available which changes the algorithm for calculation of the SQE fair share of memory for a query. With this change, when calculating the fair share of memory, the optimizer will in most cases use average active in the pool, instead of the 10% floor. The revised algorithm is:

```
If query degree = *MAX, then continue to use the entire pool.
Otherwise: SQE fair share of memory = Memory Pool Size /
"Derived_Activity"
```

If the value of average activity in the pool is at least 5, then the value of "Derived_Activity" will be the same as the average activity in the pool. If the value of average activity is less than 5 but greater than 10% of Max Active, then this value will be 5, otherwise it will be 10% of the Max Active.

1. CQE fair share of memory = Memory Pool Size / Max Active
   - If Max Active is only 1, then use 1/2 of the pool size
   - The minimum share is 100 KB
2. SQE fair share of memory = Memory Pool Size/ min(Max Active, max(Average Active Used, 5))
   The Average Active Used is the larger of the following two:
   - 10% of the Max Active specified for the memory pool
   - Average active in the pool* as reported back by the Storage Management

   ** Average active in the pool is defined as:*
   - *15 minute rolling average number of users in the pool when paging option is set to *CALC*
   - *The no. of unique users in the pool in the last 2 seconds when paging option is set to *FIXED*
3. If a job's query degree is set to *MAX, then the fair share of memory value for both CQE and SQE is equal to the entire pool size

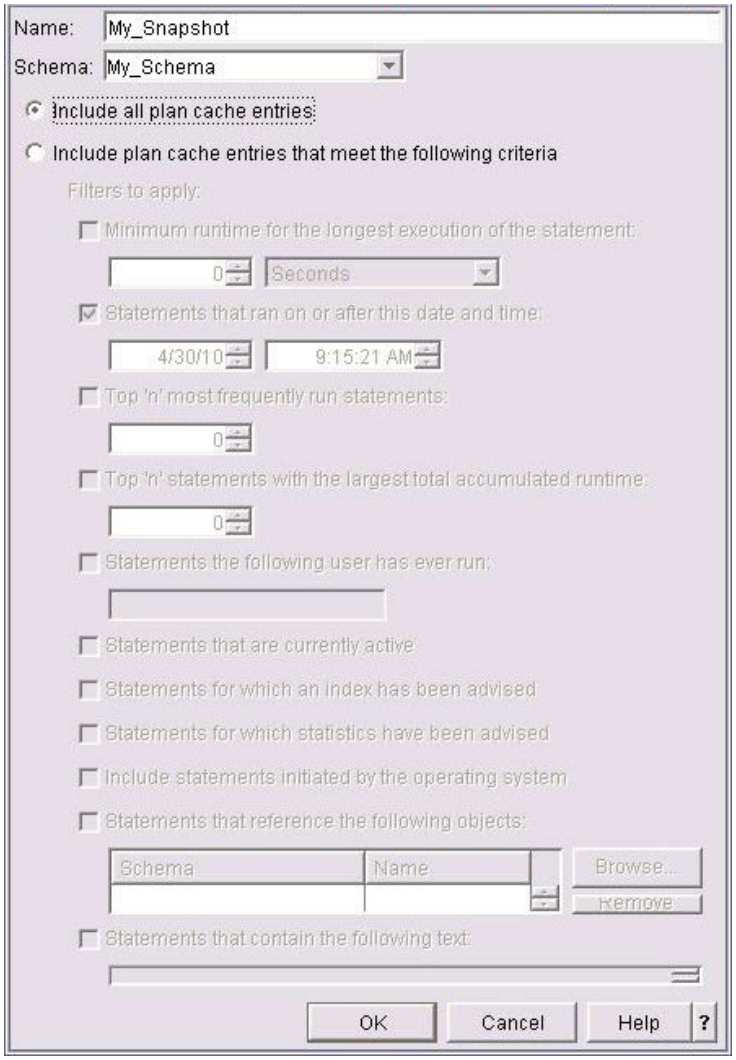
As you can see, Max Active has a strong influence on the share of memory available to a query. The share of memory available, in turn, can strongly influence what methods the query optimizer will choose when creating an access plan for that query. Also, when the share of available memory is high (as a result of smaller Max Active value), the optimizer has the freedom to pick any access method, whether it aggressively uses memory or not, in order to build the most efficient access plan. A low fair share of memory setting results in the optimizer avoiding those methods that heavily utilize memory.

## Performance investigation of the long running query

Now, let's discuss our specific SQL performance situation. The customer complained that this query used to complete in about 10 minutes, but now it was taking hours to complete. Hoping that this was a query processed by SQE, I looked for the long-running SQL statement with the

SQL Plan Cache tool. To do this, I used System i Navigator to connect to the customer's IBM i. I then clicked on Databases icon in the navigation tree and then clicked on the name of the local database to access the DB2 performance tools. After that, I right-clicked on the SQL Plan Cache Snapshot object and selected the New->Snapshot task. This action resulted in the dialog window in **Figure 1** being displayed. I took the default parameters and provided the name of My_SnapShot for the snapshot name. Creation of a plan cache snapshot allowed me the freedom of conducting my analysis at a later time - even on a different IBM i system. Note that the SQL Plan Cache only stores access plans of SQL statements processed by SQE. If this was a query run by CQE, I would have needed to collect database monitor traces using the SQL Performance Monitor tool.

## Figure 1: Creation of SQL Plan Cache

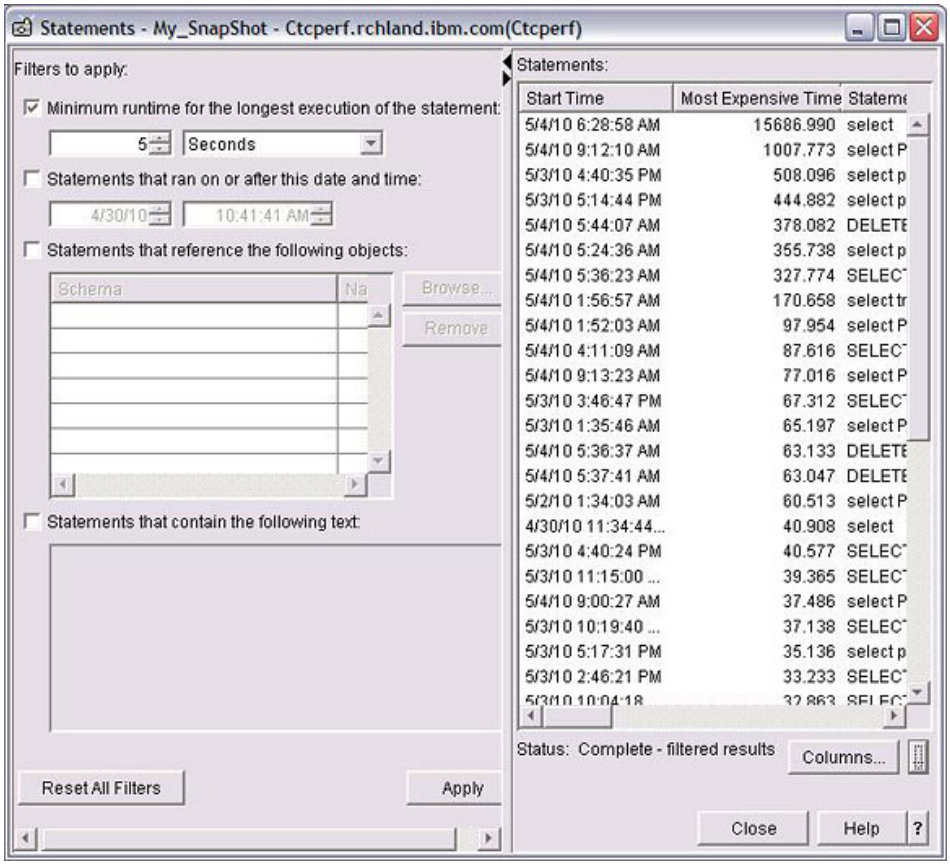



## Identification and analysis

To access the snapshot I just created, I clicked on the SQL Plan Cache Snapshots object. I then right-clicked on my saved snapshot, My_SnapShot, and selected the Show Statements task. In the dialog window in **Figure 2**, I provided a selection filter by specifying a value of 5 seconds for the "minimum run time of the longest execution of the statement" and then hit the Apply button.

This action gave me a list of the SQL statements in **Figure 2** which had at least one execution of 5 seconds or more, in the descending order of the most expensive run. Our query of interest appeared at the top of this list, and it had a run time of 15,686 seconds (over 4 hrs). To perform detailed analysis of this statement, I right-clicked on the statement to launch the Visual Explain tool. The Visual Explain tool provided me a graphical representation of the access plan that was used for this query. I reviewed the SQL statement, which was displayed in the lower panel of the Visual Explain window. The statement was a complex statement joining together 8 tables with selection predicates across multiple tables.

## Figure 2: Top statements with longest execution time of at least 5 seconds



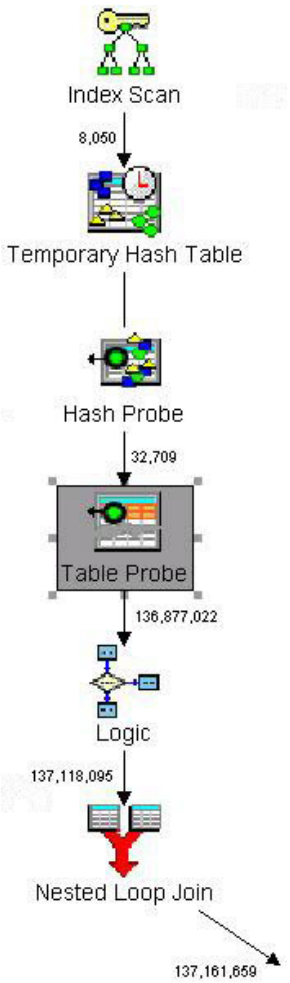

From the View tab, I selected Estimated Processing Time for the Arrow Labels, and then clicked on 'Highlight Expensive Icons" - "Estimated processing time". This action highlighted a Table Probe method for of one of the tables (call it T1). The estimated processing time for this table accounted for more than 98% of the estimated run time of the entire query (displayed under final results). I clicked on this table and reviewed information pertaining to its plan from the right hand panel of the Visual Explain. This table had more than 2 million rows, and its table probe was estimated to have almost 246 Million IO requests.

**Figure 3** shows a partial Visual Explain output which focuses on the access of table T1. Notice that the DB2 optimizer chose a Hash Table probe access method. A Hash Table probe access method is generally considered by the optimizer when determining implementation for a secondary table of a join. It requires creation of a temporary hash table with the key columns that match the

equal selection or join criteria for the underlying table. The Hash Table probe allows the optimizer to choose the most efficient implementation to select rows from the underlying table without regard for any join criteria. The hash tables are constructed with the goal that the majority of the hash table will remain resident within main memory, and the I/O associated with a hash probe will be minimal. If the hash table can be populated with all necessary columns from the underlying table, no additional table probe is required to finish processing this table.

There were 4 join and selection columns of interest for this table. For the optimal join performance of this table, the optimizer should have created a temporary hash table with all 4 columns for this step. I clicked on the Temporary Hash Table Icon to review the columns selected for creation of this hash table. I noticed that the optimizer created and populated the temporary hash table with just one column (one of the join columns). I then clicked on the Table Probe icon to review the columns accessed from the underlying table T1. The details for this method showed that the data for the 4 required columns was being retrieved by probing this table with the data retrieved from hash probe of the temporary hash table.

## Figure 3: Partial Visual Explain showing access method used for the Table T1




Creation of a single column hash table and subsequent table probe had turned out to be very expensive, because it required excessive random IOs during the table probe. So, why didn't the

optimizer create the temporary hash table of all 4 columns? To find an answer to this question, let's examine the following Environment Information for the SQL Statement:

### Environment Information for the SQL Statement

```
Memory Pool Size:                  2,454,335,488
Share of Memory Available (bytes): 81,811,184
Average Active Used:               30
Memory Active in the Pool:         300
Average Active in the Pool:        9
```

This information was obtained by clicking on the Final Result icon. The query ran in a pool of 2,454 MB, which had a Max Active value of 300 (Memory Active in the Pool). The Average Active Used was calculated as 30 (10% of 300), and therefore the optimizer determined that the fair share of the memory available to this query was 81 MB (2454/ 30). The optimizer must have determined that 81 MB was not enough room for a hash table containing all 4 columns. As a result, the optimizer decided to create a smaller hash table containing only the join column from table T1, which could fit in the available share of the memory.

## How the lowering of the Max Active helped?

From my above mentioned analysis, I was convinced that if this query had a larger share of available memory, the optimizer would chose a better and more optimized access plan for the table T1. To increase the share of the available memory, we had two choices:

1. Increase the overall memory pool size
2. Reduce the Max Activity level of the memory pool

We opted for the second choice, and lowered the Max Activity setting for the pool to 100 (from 300 before). We ran the SQL query again, and reviewed the new access plan with the help of Visual Explain. The environment information for the same SQL statement showed the fair share of memory value to be about 245 MB. The new environment information for the SQL statement was the following:

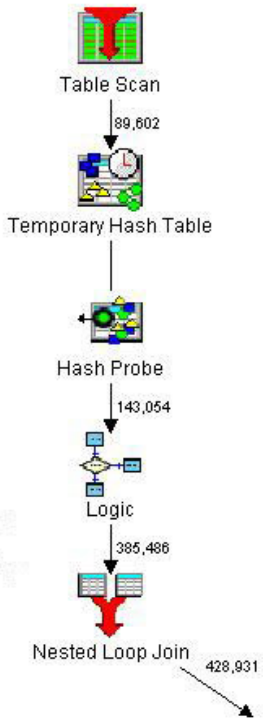
### Environment Information for the SQL Statement after the change

```
Memory Pool Size:                  2,454,335,488
Share of Memory Available (bytes): 245,433,548
Average Active Used:               10
Memory Active in the Pool:         100
Average Active in the Pool:        9
```

This time the optimizer determined that there was enough available memory for a hash table that contained all 4 columns from table T1. The size of the temporary hash table of all 4 columns was determined to be about 235 MB (this information was obtained from under the Hash Table Size panel of Visual Explain by clicking on the Hash Probe icon), which fit well in the share of the available memory. The optimizer created a revised access plan for accessing data from the table T1 **(Figure 4)**. For the new plan, a temporary hash table containing all 4 columns was created by doing a Table Scan of T1 (instead of the Index Scan used in previous run), followed by a Hash Probe. Since all of the required columns could be found in the hash table, there was no need for

a Table Probe of the underlying table. Eliminating the Table Probe operation heavily reduced the number of IO operations, and the query completed in approximately 720 seconds – A performance increase of more than 20X!

### Figure 4: Partial Visual Explain showing access method used for table T1 after lowering the Max Active value




## Conclusions and recommendations

Max Active value for a pool can have a strong influence on the query optimizer. A high Max Active value for a pool can potentially cause queries to run slow and therefore for optimal and consistent query performance, this value should not be allowed to go too high for a memory pool running queries. On the other hand, a low Max Active value may negatively impact applications which spawn multiple threads (such as Java-based applications), or for a pool where very large number of concurrent jobs are expected to run. This is because a low activity level may cause some of those threads or jobs to wait for execution. Due to this conflicting requirement, it is not advisable to share memory pool with heavily-threaded applications with that of the database server jobs (such as QZDASOINIT and QSQSRVR). In these situations, you should consider having the applications run in a separate memory pool.

## Resources

### Learn

- Check out IBM DB2 for i on developerWorks for more information.

- Read Feature: What's New: DB2 for i 7.1 to learn about other DB2 for i 7.1 enhancements.
- Learn more about Information Management at the developerWorks Information Management zone. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with developerWorks technical events and webcasts.
- Follow developerWorks on Twitter.
- Learn more about DB2 for i performance tuning from the Redbook OnDemand SQL Performance Analysis Simplified on DB2 for i5/OS in V5R4 (SG24-7326)
- Contact IBM Systems Lab Services and Training for performance consulting and training

## Discuss

- Participate in the discussion forum.
- Check out the developerWorks blogs and get involved in the developerWorks community.