It's Time to Re-think Your DB2 Buffer Pool Strategy

Martin Hubel, Martin Hubel Consulting Inc.

Abstract

As applications grow, buffer pools play an increasingly important part in reducing physical I/O, elapsed time and CPU time. Many organizations have not spent the time to optimize this important resource.

This document provides a series of buffer pool tuning ideas and techniques used to reduce the resources used by DB2 for processing I/O. The focus is several real world tuning scenarios for a variety of environments, from production and ERP applications to data warehouses. Examples from customer sites are used to illustrate the points. After physical I/O is reduced, it is also important to find ways to reduce the amount of data required by applications. This is discussed briefly at the end of the paper.

Introduction

New applications are often "strategic", and they require a significant effort to be successful. This includes the need to tune DB2 buffer pools. Even after the initial production implementation, growth should be expected and the performance characteristics will change. The tuning opportunities available, including buffer pool tuning, will change over time.

Buffer pool tuning is a major method for tuning applications. It is generally done in the production environment where the workload to be tuned resides. This diagram shows where logical I/O (GETPAGE) and physical I/O are performed.
Starting in DB2 V3, DB2 can use hiper pools to back the virtual buffer pools. This allows additional memory in expanded storage to be allocated to buffer pools beyond the 1.6 GB limit on the virtual pools. Expanded storage is used as a copy of the virtual buffer pools. It is updated by DB2, and it will provide performance benefits if a page is found there and a physical read is avoided.

Figure 1 shows where logical I/O (GETPAGE) and physical I/O is performed.

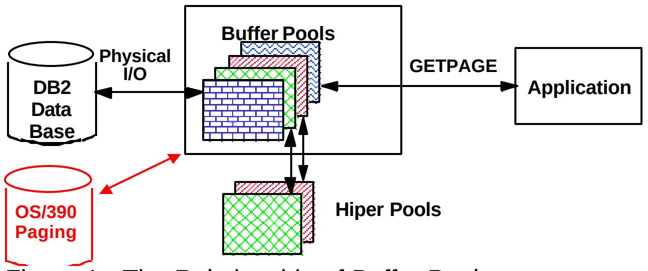

Figure 1. The Relationship of Buffer Pools

Hiper pools are slower than reading directly from the virtual pools. When hiper pools were introduced, expanded storage was cheaper than central storage. With the newer processors, all memory is the same and the system programmer decides how much will be central or expanded. The central storage should be made larger, usually reaching the current maximum of two gigabytes on an LPAR, and the expanded storage smaller. Expanded storage must not be set to zero, as OS/390 and other processes, such as some sorts, need it.

Buffer pool sizing must consider other demands of real storage. If there is insufficient real memory to back up the specifications for buffer pools, OS/390 paging will occur. This means that the entire system performance will be negatively affected. If this has occurred due to DB2 buffer allocation, the buffer pools should be reduced immediately.

DB2 has 80 buffer pools that can be used to hold table spaces and indexes. There are 50 pools for objects with a 4K page size. This includes both table spaces and indexes. The other page sizes, 8K, 16K, and 32K, are for table spaces only and have 10 buffer pools for each size. The fact that DB2 has fifty 4K buffer pools leads one to think that splitting DB2 objects into separate buffer pools is expected. .

In DB2, there are 2 basic types of I/O: logical and physical. Physical I/O is considered bad because it uses CPU resources and delays work until it completes. The delays are a result of the need to retrieve the data from a physical device before it is returned to the application.

Logical I/Os are simply application requests for data. In DB2 terminology, these are called GETPAGEs. Through the use of areas of memory, called buffer pools, DB2 can substantially reduce CPU, I/O, and elapsed time. This is reflected by reduced resource consumption and improved user productivity.
It is desirable for a logical I/O to not require a physical I/O. If the data resides in the buffer pool(s), the request is much cheaper than waiting for physical I/O to complete. Updates to data are made in the buffer pools, and these changes are written back to DASD when threshold values are reached or at system checkpoint time. The buffer pools in central storage are also called virtual buffer pools.

Buffer Pool Thresholds

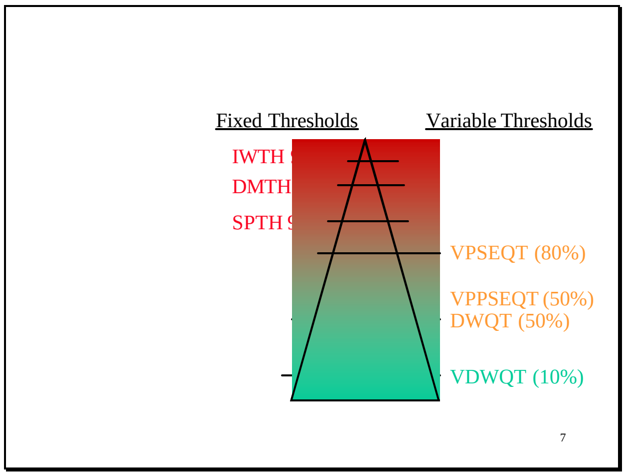

Figure 2. Buffer Pool Thresholds

DB2 uses thresholds to control buffer pool processes. These processes control the type of I/O operations that are performed by DB2, such as asynchronous read or write operations, and the limiting of buffer usage by these operations if buffer pool thresholds are exceeded. Buffer pool utilization is computed as a ratio of:

(Updated Pages + Pages In-Use) / BP Size

The fixed thresholds are as follows:

- SPTH - Sequential Prefetch Threshold: At 90%, DB2 will inhibit sequential prefetch.
- DMTH - Data Manager Threshold: At 95%, DB2 will perform all GETPAGE & RELEASE at row level. CPU usage is very high.
- IWTH - Immediate Write Threshold: At 97.5%, any update causes an immediate write, synchronous with request (1 I/O per update). This threshold can be reached without SPTH and DMTH being hit, and in this case it is not a problem.

The variable thresholds can be tuned, and are as follows:

- DWQT - Deferred write. Percent of updated/in-use pages before asynchronous writes are scheduled.
- VDWQT - Vertical deferred write. Like DWQT, but for a single dataset. May also be called VWQT.
- VPSEQT - Sequential steal threshold. % of pages occupied by sequentially accessed pages
- VPPSEQT - Parallel sequential threshold. % of the buffer used for parallel operations. It is expressed as a percentage of VPSEQT.
- HPSEQT - Hiperpool sequential steal threshold. %of hiperpool used by sequentially accessed pages. This is the only threshold for hiperpools.

The fixed thresholds severely limit the asynchronous buffer pool activity, and cause additional CPU time to be spent by DB2 performing I/O. It is best to plan such that SPTH and DMTH are never reached. This can be done in two ways: having sufficient buffer sizes for your workload, and setting variable thresholds appropriately. Various recommendations regarding the variable thresholds follow.

The Vertical Deferred Write Threshold (VDWQT) should be used to control asynchronous write operations. The intent of setting the VDWQT to a small value is to have a regular asynchronous write of updates to DASD. A page that is referenced frequently will remain in the buffer even if updates to it are written back to DASD. Setting VDWQT to 0 will start a write operation of 32 pages when 40 pages have been updated.

Certain DB2 operations may cause an immediate write. For example, a page that has remain in the buffer pool for two checkpoints will be written to DASD. Provided SPTH and DMTH are both zero, a small number of hits for IWTH does not indicate a problem.
The work table spaces in DSNDB07 should be placed in their own buffer pool. Typically, this is only 4 to 6 datasets. Due to the small number of datasets and the desire to avoid physical I/O, the DWQT and

VDWQT for this buffer pool can be set higher. For buffer pools whose intent is random I/O, the sequential thresholds can be set lower to avoid dominance by sequential operations in these pools.

Types of Physical I/O

Physical I/O includes many additional things such as rotational delay for the disk drive and contention for the device and channels. A good access time for a single page from disk is measured in milliseconds. A GETPAGE satisfied from the buffer pool is measured in microseconds.

There are two major types of I/O performed by DB2. The first type is random I/O. This type of I/O means that a given piece of data is read by DB2 using an index to find the appropriate page and retrieve the data required. Random I/O is efficient and is even better when no physical I/O is required to satisfy the request.

The second type of I/O is sequential I/O. As its name implies, sequential I/O requires that all or part of a table space or index be read sequentially to find the data required by an application. This usually means a lot of I/O and processing by DB2 to find the result requested. Sequential I/O can lower the possibility of finding a page in the buffer for another request. Prefetch I/O takes longer than random I/O, but it returns more pages per I/O.

For 4K buffer pools that are 1,000 pages or larger, DB2 will read 32 pages in one prefetch I/O for SQL and 64 pages for utilities. For 32K buffer pools, 8 times fewer buffers are used than for 4K pools. Prefetch works so well that two additional types of prefetch were added to the original sequential prefetch. These are list prefetch and sequential detection or dynamic prefetch. Prefetch causes many processes that might be bound by I/O to become CPU bound. DB2's decision to use prefetch can happen at bind or at execution time.

Pure sequential prefetch means that the optimizer at bind time has chosen to use prefetch. Both pure sequential and list prefetch are shown on the Explain output. Prefetch can be determined at BIND time or dynamically at execution time.

List prefetch is like skip sequential processing: a list of Row IDs is built from one or more indexes and sorted before the data is retrieved in the order it is stored on DASD. Up to 32 data pages can be read (based on prefetch quantity) before the first row is returned to the application program. DB2 triggers sequential prefetch for the initial set of pages read. This may adversely affect some "BROWSE" transactions, as the entire list of RIDs must be built and sorted before the first row is returned to the application.

When list prefetch is used, the result is returned in increasing RID order, which is the order of the data on DASD. This would be in the order of an organized clustering index. Use ORDER BY clause if the order of the result is important.

The processing for list prefetch is as follows:

1. Get list of row IDs (RIDs) from each index
2. Sort RIDs
3. Perform union / intersection on RIDs if multiple indexes are used. This is based on the condition in the WHERE clause. Union is done for OR predicates, and intersection is done for AND.
4. Perform List Prefetch for pages in RID list to retrieve data rows

As a further improvement to sequential prefetch, IBM introduced a dynamic prefetch in DB2 V2R3. Dynamic prefetch, or sequential detection, is a run time decision by DB2 if the processing finds a sequential pattern in the data. It is turned on if 5 of the last 8 pages are within half of the prefetch quantity of each other. Monitoring continues, and if the pattern changes to a point where sequential prefetch is not helping, it is turned off. This can happen multiple times in one execution.

Note that any type of prefetch may indicate a poor access path for on-line or ad hoc processing. Prefetch will also reduce the space available in the buffer pool for randomly accessed pages.

Hit Ratios

Proper buffer pool tuning can improve application response times by reducing I/O wait times and the CPU resources required to perform physical I/O. Key measurements, called the system hit ratio and the application hit ratio, are determined by the number of requests for the page (getpage) and the number of times a DB2 page is found in the buffer pool.

The application hit ratio is typically used by on-line monitors, and is computed as:

getpages / read I/Os or (getpages-read I/Os) / getpages

This formula is useful only for application performance where almost all access is random. This ratio could also be converted to a percentage. Most application workloads have both random and sequential access to data.

The system hit ratio provides a more accurate measurement:

(getpages-pages read) / getpages

The system hit ratio is a more meaningful measurement because it shows the request for pages and the number of pages that must be read from DASD to satisfy the request. Asynchronous I/O via sequential prefetch can have a negative impact on buffer pool efficiency. It is possible to have a relatively high application hit ratio while also having a low system hit ratio.

Typical Buffer Pool Starting Points

Many organizations start out using four or fewer buffer pools. As an example of a four pool starting configuration, BP0 is used for the catalog and directory, the temporary work database is in BP1, and the table spaces and indexes are in BP2 and BP3 respectively. For the purposes of this document, this is a reasonable starting point, and usually presents great tuning opportunities. The minimum size for any pool should be 1,000 pages. Below this size, the prefetch quantity is halved.

It is critical to have sufficient real storage available to avoid OS/390 paging. As a rough starting point at one site, we set BP0 at 5,000, BP2 at 15,000, and BP1, BP3, BP4 and BP5 at 10,000 buffers each in a ERP only environment. Using these values, we were able to reduce the amount of physical I/O to a point where performance became satisfactory. Additional buffers were added as further opportunities for improvement were identified.

A Methodology For Buffer Pool Tuning

The objective of buffer pool tuning is to increase the residency of high-use pages in the pool. To obtain the best tuning results, it is important to tune for the peak periods of application use. Large buffer pool sizes combined with proper allocation of objects to buffer pools can make dramatic improvements to application performance.

The immediate benefits of buffer pool tuning are improved on-line response times and batch run times realized by users of the DB2 system, improved user productivity, greater system throughput, reduced CPU utilization, reduced I/O workloads, and the optimization of DB2 memory resources.

Separate By Type of Object

The main methods to improve performance are to separate table space from indexes, and then to separate by the type of I/O being performed against objects. Large objects (LOB) should also be placed in a separate pool, as should high-use read only objects.

It is important to identify the high-use tables and indexes in your application. You can identify these from explain output, tuning software, or your knowledge of application processes.

In general, a larger buffer pool is better, but an over-allocated pool might be keeping memory from another pool that would receive more benefit. Also, one poorly performing large object can negatively impact performance of other objects.

Separate By Type of Access

It is important to identify I/O activity, type and rate of access, and reference patterns for each DB2 object. Objects having poor access methods (such as frequent large scans) should be identified and steps taken in the application to improve performance or be moved to another pool to reduce the impact to on-line transactions.

Random I/O is efficient and is even better when no physical I/O is required to satisfy the request. A large buffer pool will allow high-use pages to remain in the buffer pool. Sequential I/O lowers the possibility of finding a page in the buffer for another request, and often pushes other pages out of the buffer. Even if a large buffer size is used, there is a lower chance of a page still being resident. Separating sequentially accessed objects into a buffer pool away from randomly accessed objects will improve the performance of the random objects. Sequential objects should not be expected to perform well regardless of the buffer pool size. A pool of 1,000 to 3,000 buffers should be adequate depending on the number of objects and how heavy the workload is. A larger number may be required.

Of course, life gets interesting if an object is accessed both sequentially and randomly. Perhaps index tuning to modify the sequential access paths will help. Also, it may be possible to modify buffer pool allocations at certain times of the day to improve DB2 performance when other workloads will not be affected.

Very large tables are a special case. As an example, a very large data warehouse has high activity against some of its atomic (fact) tables with 3 tables having over 4 million pages each. All access to these tables is through a unique index, matching on all index columns. Any other access is measured in minutes, rather than seconds, and is quickly fixed. Based on the size of these tables, there is little chance of a page being found in the buffer pool. Increasing the size of the buffer pool is unlikely to improve the hit ratio for these table spaces. The size of the table spaces means that it should be isolated from other randomly accessed table spaces. Of course, they could also be treated as a poor performing asynchronous objects and placed in a sequential pool. A pool of between 1000 to 3000 pages is sufficient.

A Very Large Example

A client was concerned about the effect of PeopleSoft Financials and HRMS on other production applications, so it was placed in its own LPAR. This LPAR was given 2 GB central and 2 GB expanded storage. Table 1 shows the initial allocation. While the size of the pools is very large, the general philosophy is typical of how objects should be allocated to DB2 buffer pools.

Using fairly large sizes for the EDM, RID and sort pools still mean that the maximum 1.6 GB could be allocated to buffer pools. My initial reaction was to specify about 900MB for 4K BP and hold some memory in reserve to use as the data volumes grow, but a “use it or lose it” philosophy is present in this shop. We changed VDWQT to 1 on all pools except BP7, where it was set to 50%. DWQT was set to 60% on BP7. VPSEQT was set to 30 on random pools and 100 on sequential pools including DSNDB07.

Random and general pools should have mainly random access to a few pages per request. Large pools will allow more pages to be found in memory and avoid physical I/O. Most physical I/O will be synchronous rather than asynchronous on the large pools. As a slightly phased implementation is being done, no physical reads are expected after the initial load in the pools.

Buffer Pool Name	Initial Size # buffers	Proposed Uses
BP0	25,000	DB2 Catalog and Directory
BP1	60,000	Indexes - General Pool plus 100,000 HP
BP2	40,000	Indexes – High use reference (synchronous)
BP3	5,000	Indexes – poor performers (asynchronous)
BP4	60,000	Table spaces – General Pool plus 100,000 HP
BP5	60,000	Table spaces – high use reference (synchronous)
BP6	5,000	Table spaces – poor performers (asynchronous)
BP7	30,000	Work data sets only
BP8	15,000	Special use tuning – indexes. For characterizing a single index
BP9	15,000	Special use tuning – table spaces. For characterizing a single TS
BP32K	300	Table spaces – 32K General
BP32K1	300	Table spaces – 32K Special
Total	315,600	1.26 GB for 4K pages, 9.6 MB for 32K pages

Table 1. Large BP Allocation for separate LPAR

Poor performers will have primarily sequential access. This type of access involves the sequential prefetch that can quickly overlay existing pages in the buffer pool. Large amounts of memory would not improve its performance. Most physical I/O is asynchronous rather than synchronous.

To date, data and transaction volumes are light, and system hit ratios are at least 98% on all pools. The pools are over allocated until production volumes increase.

Buffer Pool Monitoring

Each monitor has its strengths and weaknesses, and you should learn the features of your monitor for each tuning scenario. This is particularly true for buffer pool tuning.
For computed values, it is important to check computations to ensure they are correct. For example, one monitor did not compute getpages/read I/O correctly when the number of getpages went to 9 digits. Another product did not include the asynchronous read I/Os in the application hit ratio.

Another on-line monitor computes the system hit ratio, but it counts hiperpool reads as misses rather than hits. With this product, it is possible to easily view the formula for the hit ratio and correct it if desired.

Some products are easily customized, which makes it easy to add items like the system hit ratio to reports. Another important aspect of customization is the removal of fields that are not important. For example, buffer pool reports for a read-only data warehouse do not need the section on buffer pool writes. The only way to correctly tune buffer pools is to use object level information. Very few products supply any information at this level.

Data Warehouse Example

This example is a data warehouse implemented on DB2 for OS/390. The amount of data in tables and indexes is 450 GB with the data compressed. Some tuning has been done, and performance has improved as shown in Table 2. There are still a number of tuning opportunities available as indicated on the performance report on the following pages.

Another problem that is hard to overcome is the combining of pools. The subsystem has too many buffer pools, but there is a reluctance to take steps to reduce the number as performance is already so much better than before. It is best to avoid defining more buffer pools rather than worrying about combining pools later.

BP Name	VP Size July	HP Size July	VP Size October	HP Size October
BP0	3,000	7,000	3,000	7,000
BP1	10,000	10,000	10,000	10,000
BP2	6,000	0	6,000	0
BP3	10,000	10,000	20,000	40,000
BP4	5,000	0	5,000	0
BP5	4,000	0	4,000	0
BP6	5,000	10,000	20,000	40,000
BP14	2,000	0	0	0
BP15	1,000	0	0	0
BP16	2,000	3,000	0	0
BP17	1,000	0	0	0
BP18	15,000	40,000	22,000	48,000
BP19	3,000	6,000	3,000	6,000
BP20	5,000	5,000	5,000	5,000
BP21	5,000	5,000	5,000	5,000
BP22	5,000	5,000	5,000	5,000
Total	82,000	101,000	108,000	166,000

Table 2. Data Warehouse Example

```
REPORT(SHBUFFUG/D)   --    BUFFER POOL USAGE GRAPHS    --      02/24/99 14:00:49
Group :          Member :           SSID : DB2W(4.1) Loc : DB2W
    Begin record : 02/24/99 07:00:01        Elapsed time : 00:20:00
    End record   : 02/24/99 00:20:00      Total Getpages : 1144070K
    Vpool   Hit ratio  Getpages   Percent of Total Getpage Activity in Subsystem
    -----   ---------  --------  |---------------------+----------------------|
 +  BP3         15%     124483K  |>>>>>                                       |
                                 |---------------------+----------------------|
    Elapsed time : 07:00:29
    Sample time       Ratio  Getpages   SyncI/O    SyncH/S   AsyncI/O  AsyncH/S
    02/24/99 07:00:01    16   107075K    920939     178030   85028658   3952040
    02/24/99 07:10:00  -131    529746      1692        503    1015312    204870
    02/24/99 07:20:00  -248    718979       705        186    2289721    209208
 +  BP19        98%    26212453  |>                                           |
                                 |---------------------+----------------------|
    Elapsed time : 07:00:29
    Sample time       Ratio  Getpages   SyncI/O    SyncH/S   AsyncI/O  AsyncH/S
    02/24/99 07:00:01    98  20465303     36923       6018     180243    181403
    02/24/99 07:10:00    99    242455       160          0       2138         0
    02/24/99 07:20:00   100    122452         1          2        179        66
```

Figure 3. Abridged Screen Shot showing BP Utilization for Data Warehouse

Figure 3 is an excerpt of a BMC Activity Monitor report from the data warehouse. The total activity for the applications is quite high at 1.1 billion getpages over a 2 day period. This report is better than many monitor reports because it shows the pages read rather than just the number of I/O operations. BP19 is used for some high use indexes, and is performing very well. This is in spite of the fact that this BMC report counts a hiperpool I/O as a miss rather than a hit. Pages found in the hiperpool are normally counted as a hit as some tuning action was taken to achieve that result.

BP3 is performing poorly. The negative hit ratio means that some pages are being read asynchronously, but are not being processed before they are replaced in the pool. They have to be read again. There could also be a large amount of dynamic prefetch. After further investigation, it might be appropriate to separate poorer performing objects from the other objects in this pool.

Buffer Pool Tuning Mistakes

The following examples show some typical mistakes made by organizations as they tune buffer pools. The mistakes are often the result of a combination of misunderstanding how DB2 uses buffer pools and a neglect of proper buffer pool monitoring.

Mistake #1: Hitting Thresholds

The DB2 subsystem in Figure 4 uses BP7 for its work table spaces. Based on the small number of table spaces in DSNDB07, it is appropriate to increase VDWQT to a higher number. DWQT was also increased. The default values for VDWQT and DWQT are 10% and 50% respectively. As in the case of the work buffer pool, it makes sense to raise VDWQT to allow more sorts and temporary tables to be stored in the buffer and avoid physical I/O.

```
SET    : TMONDB2        CURRENT BP UTILIZATION DETAIL         DATE: 06/16/99
DB2    : DP01                                                 TIME: 17:16:41
               INTERVAL FOR: 06/06/99  11:18:21 THRU 17:16:34
BPID: BP4K07
                           BUFFER POOL UTILIZATION
UTILIZATION PCT       :     32.2        WRITE ENGINE UNAVAILABLE  :       8
# BUFFERS IN POOL     :     5000        DATASET OPENS             :       5
ALLOCATED BUFFERS     :     1613        MIGRATED DATASETS         :       0
                                        DATASET RECALL TIMEOUTS   :       0
                                        PREFETCH DISABLED
THRESHOLD                  HITS  PCT.      NO READ ENGINE         :       0
SEQUENTIAL                   :      N/A     90%          QUANTITY 0                  :
24603635VERTICAL  DEFERRED  WRITE:    16049      60%      ALTER  BP  EXPANDS/CONTRACTS:
0
DEFERRED WRITE         :  10071   80%
PREFETCH               :   2486   90%
DATA MANAGER CRITICAL  :    704   95%
IMMEDIATE WRITE        :     10   97%
EXPANSIONS             :      0  100%
    FAILED - MAXLIMIT  :      0
    FAILED - VIRTUAL   :      0
```

Figure 4. Buffer Pool Thresholds Hit for Work Buffer Pool

Problems occurred due to the aggressive settings of VDWQT and DWQT to 60% and 80%. Heavy periods of usage did not allow DB2 to act on these thresholds and the critical thresholds of SPTH and DMTH were reached repeatedly.

In this situation, it would be appropriate to decrease these thresholds to 50% and 60%, or to increase the size of the buffer pool. If applications do a lot of sorts or joins, a larger buffer pool will help. Also, the QUANTITY 0 indicates that prefetch is being turned off. This could be due to the small size of the buffer pool. There is also a V5 APAR PQ32018 pertaining to excessive synch I/O for sort work files.

Mistake #2: Ignoring Tuning in Test Environments

Many organizations spend little or none of their tuning efforts on development, and some are even unaware of the resources used. Major development projects can use large amounts of resources. If tuning is not done, project deadlines could be missed, due to lost productivity. This could also result in increased consultant costs if they are used.

Another aspect of tuning before production is access path tuning. Access path selection takes buffer pool size and CPU speed into consideration. A stress testing environment should have comparable buffer pool allocations to production.

The following two screen shots the effect of a poorly tuned test environment. As it turns out, the PeopleSoft project team was doing more I/O than all but 1 production DB2 subsystem. Figures 5 and 6 from the Landmark DB2 monitor have to be read carefully as they show the getpage activity and the number of I/Os. They do not show the number of pages read from DASD, although this information is available by drilling down.

```
*SET    : TMONDB2          CURRENT BUFFER POOL SUMMARY         DATE: 05/10/99*
 DB2    : DT03                                                 TIME: 16:36:43
                 INTERVAL FOR: 05/02/99  22:29:00 THRU 16:28:48

                                                                LOCAL  TOTAL
  BP     % USE  GETPAGE  SYNCREAD  ASYNREAD  SYNCWRIT   HP IO   GBPIO  GBPIO
  BP4K00   0.5 53709761    846087    814867      5641       0       0      0
  BP4K01   1.2 1118.50M    739095    537287      2690       0       0      0
  BP4K02  12.8  643.86M   1096066    172791      9246       0       0      0
  BP4K04   0.0        0         0         0         0       0       0      0
  BP4K05   0.0        0         0         0         0       0       0      0
  BP4K07  17.6 16370775     43559    550569         4       0       0      0
  BP32K0   2.0   230067      4742     15610        39       0       0      0
```

Figure 5. Buffer Pool Summary from the Landmark Monitor

```
*SET    : TMONDB2        CURRENT BP UTILIZATION DETAIL         DATE: 05/10/99
 DB2    : DT03                                                 TIME: 16:29:17

                 INTERVAL FOR: 05/02/99  22:29:00 THRU 16:28:48
 BPID: BP4K00
                              BUFFER POOL UTILIZATION
 UTILIZATION PCT       :      0.5       WRITE ENGINE UNAVAILABLE  :        0
 # BUFFERS IN POOL     :      750       DATASET OPENS             :    17464
 ALLOCATED BUFFERS     :       25       MIGRATED DATASETS         :        0
                                        DATASET RECALL TIMEOUTS   :        0
                                        PREFETCH DISABLED
 THRESHOLD                  HITS  PCT.     NO READ ENGINE         :        0
 SEQUENTIAL             :    N/A   80%     QUANTITY 0             :        0
 VERTICAL DEFERRED WRITE:  53462   10%  ALTER BP EXPANDS/CONTRACTS:        0
 DEFERRED WRITE         :   4471   50%
 PREFETCH               :  11617   90%
 DATA MANAGER CRITICAL  : 728369   95%
 IMMEDIATE WRITE        :   5641   97%
 EXPANSIONS             :      0  100%
     FAILED - MAXLIMIT  :      0
     FAILED - VIRTUAL   :      0
```

Figure 6. Buffer Pool Detail for BP0 Showing Thresholds Reached

An additional PeopleSoft environment was added to the subsystem. Based on the small buffer pool size, the data manager critical threshold (DMTH) was hit an incredible number of times. The sequential prefetch threshold was also hit. As the number of buffers was under 1,000 (but above 223), the prefetch quantity was 16.

While this mistake was caused by a dismissal out of hand of the development environment, it is not suggested that an inordinate amount of time be spent on it. Based on the dynamic SQL used by the application, BP0 was increased to 5,000 and the performance was much better.

Mistake #3: Separating Buffer Pools by Application

A production system only was using 10,000 buffers. In an attempt to improve performance, the BP size was increased to 30,000 buffers and each application was moved to 2 separate pools, one for table spaces and one for indexes. The catalog and work datasets were in BP0 and BP1 respectively. There was no change in performance. There are too many buffer pools being used. This is shown in Table 3.

POOL	VP Size	POOL	VP Size
BP0	1000		
BP1	2000		
BP2	4000	BP12	2500
BP3	3000	BP13	1500
BP4	500	BP14	250
BP5	2000	BP15	800
BP6	5000	BP16	2500
BP7	2000	BP17	1000
BP8	1500	BP18	500
BP32K	250		

Table 3. Buffer Pools By Application

A few mistakes were made here. First, some applications were given so little buffers that the prefetch quantity was halved. Regardless of your best tuning efforts, there will always be some sequential access, so it is very important not to hinder efficiency in this area. A pool of less than 1,000 pages is generally useless, unless it is desired to page fix some very small objects.

Second, some applications may be busy at different times of the day. By segregating applications, it is impossible for other application to use this BP space. It is important think of memory as a shared resource. And third, no attempt was made to separate by type of I/O.

Reducing Logical I/O

Using memory and allocating large buffer pools can cover up a lot of application "sins". This can work very well in many cases. However, there are times where the problem is simply the amount of logical I/O. It is therefore necessary to reduce the application demand for data. In these cases, we need to reduce the getpages performed by the application.

These methods include index redesign, hardware data compression, maintaining good physical data organization, and rewriting SQL. The first 3 methods can be done by the DBA without a real need for application team involvement. In addition, these methods do not need the involvement of the application team. Rewriting SQL is done by the application area with the help of DBA. Rewriting SQL and application testing requires significant effort including proper change management.

Index Change Guidelines

Figure 7 represents general guidelines for tuning indexes of an existing application. Except where referential integrity is involved, indexes are easily modified. The technique to remove indexes and unnecessary index columns to get more index keys per page.

1. Do not change the primary index
2. Expect little index only access
3. Check clustering index placement
4. Remove redundant indexes
5. Remove unnecessary index columns
6. Add indexes carefully when necessary

Figure 7. Index Change Guidelines

The primary index is often the only index defined as unique on a table. Adding or removing columns in the primary index will change the uniqueness of the index key and affect the integrity of the application. Changes to the primary index should only be done with the assistance of application personnel who understand the application.

Index only access is a useful feature for on-line transaction processing where a single or very few columns are needed from a query. This feature is often overused; the result is keys that are very long and/or the result still requires more columns from the data. In either case, the index keys end up being longer than necessary, meaning that fewer index keys appear on each page and the index is larger. If it is necessary to scan the index entries, more pages will need to be scanned which would result in more GETPAGES and possibly more physical I/O. It is often cheaper to read a short index key and retrieve the desired columns from the table.

The definition of clustering varies between DBMSs. For example, clustering in Oracle involves the interleaving of table rows on the same page to help with parent-child retrieval. This is very different from the definition in DB2, where an index is chosen as clustering and the data is physically stored into the order of this index. In DB2, this means that fewer pages need to be read to return a multiple row answer set.

The importance of the clustering index to reducing I/O and elapsed time has been shown time and again. Unless you have a meaningful primary key where users might search on ranges or batch processes that process in that order, it is usually better to choose another index as clustering. Often, a mistake is made where the primary index is defined as the clustering index as well. In many cases, this provides less benefit than other possibilities as the primary key is unique and is not often used to return a range of values. Of course, there are exceptions.

Some tuners seem to add indexes for every possible access path. This causes another problem where the large number of indexes competes for buffer pool space. For example, if there are two indexes that have four or five columns each, and the first three columns are the same, it may be better to have only one index if the cardinality of the first three columns is high enough.
Some columns are added to indexes that do not help DB2 to narrow the search for the answer set or they are redundant because columns closer to the front of the index have already narrowed the search to one or two data pages. These columns can be removed from the respective indexes to allow more entries per page and consequently make the indexes smaller.

It seems sometimes that many people have heard that the columns with the higher cardinality must be in the first index column. This is true only if the search value is known and specified in each query. The first columns of an index should be those that are known, regardless of their cardinality. Of course, columns with a cardinality of 1 do not help narrow the search, and they should be removed from indexes.
I/O is reduced by properly designed indexes that hopefully stay resident in the buffer pool. Adding more indexes not only takes more CPU time during updates, but they take up more buffer pool space. This "contention" for buffer pool space can result in more physical I/O. This effect can be hard to measure, as some processes may speed up with the new index, but the overall effect to performance may be negative.

Hardware Data Compression

In the earlier releases of DB2, data compression was considered to be a trade-off between saving DASD and the use of CPU to compress and decompress the data. Unless the compression rates were sufficiently high, data compression would not provide sufficient benefit to make it a viable choice. In many cases, the cost of compression was offset by the CPU savings of doing fewer I/Os as a result of having more rows of data per page, and thus fewer pages.

The newer processors and DB2 V3 introduced a hardware data compression feature for table spaces. Indexes, and DB2 components such as the catalog and directory are not compressed. Using microcode to perform the compression and decompression, the impact on CPU time is minimal. The Lempel-Ziv compression algorithm is quite good for most types of data. It uses a compression dictionary to store the most frequently found patterns within a table space, which is similar to the popular PC program PKZIP.

Note that compression happens at the row level, meaning data is compressed in the buffer pool. Only the rows needed by the application are decompressed.

This feature can be turned on for existing tables paces using the COMPRESS YES parameter on the table space. The next LOAD or REORG will build the compression dictionary and compress the data. If the measured benefits are insufficient, the parameter can easily be changed back to COMPRESS NO and followed by another REORG. As such, there is a minimal risk when compressing a table space. If you achieve good benefits, fine. If you don't, the additional resources will be less than 5 per cent for that object until you can return it to an uncompressed state.

For the more cautious among us, the utility DSN1COMP can be used to check the effectiveness of compression to choose candidate table spaces. Many applications, particularly ERP, can achieve compression rates of 60% or more.

Physical Data Organization

It may seem obvious, but poor physical data organization can cause more logical and physical I/O to retrieve data. It decreases buffer pool efficiency and increases CPU usage. Many organizations schedule REORGs on a time basis, such as once per week or even more often for heavily used table spaces and indexes. Other organization uses a utility or product if certain key parameters pass a threshold. Examples of key thresholds are CLUSTERRATIO below 95%, LEAFDIST greater than 200, dataset extents > 10, etc.

Buffer Pool Enhancements in DB2 V6

New Page Sizes

New intermediate page sizes have been added to allow better flexibility in managing longer row sizes and better matching them to a page size. There are 10 buffer pools for each new page size as there is for 32K pages. Of course, everything is still managed by DB2 as multiples of 4K page sizes. DB2 uses an asynchronous write size of 128K. The larger the page size, the fewer pages per write I/O.

Default Buffer Pool Parameters in DSNZPARM One of the recommendations for tuning buffer pools is to always reserve BP0 for only the catalog and directory. Before V6, this could require some active management to keep vendor products and user databases from using BP0. In V6, it is possible via 2 additional parameters in DSNZPARM and on the CREATE and ALTER DATABASE statements to specify the default buffer pool for table spaces and indexes. This also reinforces the concept of placing table space and indexes into different buffer pools.

Finer Tuning of VDWQT

For very large buffer pools, a vertical deferred write threshold of 1% can still cause an I/0 spike. For example, a pool of 60,000 pages will write 600 pages, or 2.4 MB, when the threshold is hit. As mentioned earlier, a VDWQT value of 0 will write 32 pages as soon as the updated buffer count reaches 40. The V6 form of the ALTER BUFFERPOOL for VDWQT allows an absolute number of buffers to be specified as a second integer if the first integer is zero, for example, VDWQT(0,100) uses 100 buffers for VDWQT.

BP Page Stealing Algorithm

The buffer pool page stealing algorithm was always least recently used (LRU) before DB2 V6. This means that a highly used page can remain in the pool based on its continued use. There is some overhead associated with maintaining the LRU chain. First in first out (FIFO) means the oldest page will be removed first. FIFO may improve performance if the BP size allows objects to be fixed in the pool, or the objects are sufficiently large that there is little chance of a page being referenced again.

LOB Support

V6 large objects (LOB) are stored in a separate table space. The allocation of the table space and its corresponding index can be automatic if the CURRENT RULES special register is set to STD. LOB TS can optionally have logging turned off. Based on the size of LOBs, the TS should be placed in a separate buffer pool. Consider using a small pool as the size of the LOBs should make it unlikely that a LOB would remain in the buffer pool.

V6 Buffer Pools in Dataspaces

Dataspaces were introduced years ago with MVS/ESA. The main difference is data integrity. Hiperpools provide read integrity, but dataspaces guarantee it. As such, DB2 can read and write directly to dataspaces. This will allow buffer pool sizes to be larger than the current limit of 1.6 GB for all virtual buffer pools. To use dataspaces, specify VPTYPE(DATASPACE). Hiperpools cannot be used with dataspaces.

The maximum size of all dataspaces used for DB2 is 8 million pages. If all 4K pages are used, the maximum size is 32GB. If all 32K pages are used, the maximum size is 256GB.

The world of dataspaces is not currently perfect. Current hardware and operating system considerations require the use of a lookaside buffer in the DBM1 address space to boost performance. Do not expect dataspaces to perform as well as current virtual buffer pools. 64-bit addressability in Release 10 of OS/390 will increase the performance of buffer pools in dataspaces.

Conclusion

Buffer pool tuning is one of the best and easiest methods for improving DB2 performance. Tuning should be done by separating table spaces from indexes, and separating randomly accessed objects from sequentially accessed ones.

Tuning should be done initially in production where the greatest amount of data resides, and presumably where the service requirements are highest. Following production tuning, the test systems should be reviewed to ensure development productivity is satisfactory.

Finally, tuning is an ongoing process. To achieve good benefits, it is important to implement proper buffer pool monitoring procedures.

Bibliography

Joel Goldstein, “DB2 System Performance Metrics” CMG 1996, IDUG 1997