

Memory Management for Dynamic SQL

Thomas Baumann, La Mobilière (Suisse)

thomas.baumann@mobi.ch



Réunion du Guide DB2A
Jeudi 3 avril 2008
Infotel, Bagnole (93)
France

La Mobilière
Assurances & prévoyance



Dynamic queries use many memory resources to back dynamic statement caches, buffer pools etc.. 64bit systems offer a large amount of tuning opportunities, and many possibilities to waste memory. This presentation takes you from an overview of cache usage, dives into the caches to detect meaningful information, comes up with a 'memory efficiency score', and shows you how to estimate the benefit you could expect if you install 2GByte of additional memory. All with a specific focus on dynamic SQL queries.

Key Points

- Overview of dynamic SQL's usage of memory
- Queries and commands to get meaningful memory information
- Benefit from experiences at other customer's sites
- Predict the benefit of additional memory
- Memory efficiency check

How do (dynamic) queries use memory resources?

- Dynamic statement cache
- Virtual and group buffer pools
- Other

How much memory do dynamic queries use?

- Inside the dynamic global statement cache
- Workload information derived from cache inspection
- Inside the buffer

How efficient do dynamic queries use memory?

- Checklist to compare goals and measurements
- Should I decrease the statement cache in order to increase the buffer pool?

How much benefit would I have of 2GByte additional memory?

- Estimated I/O reduction time
- Estimated CPU reduction

Agenda

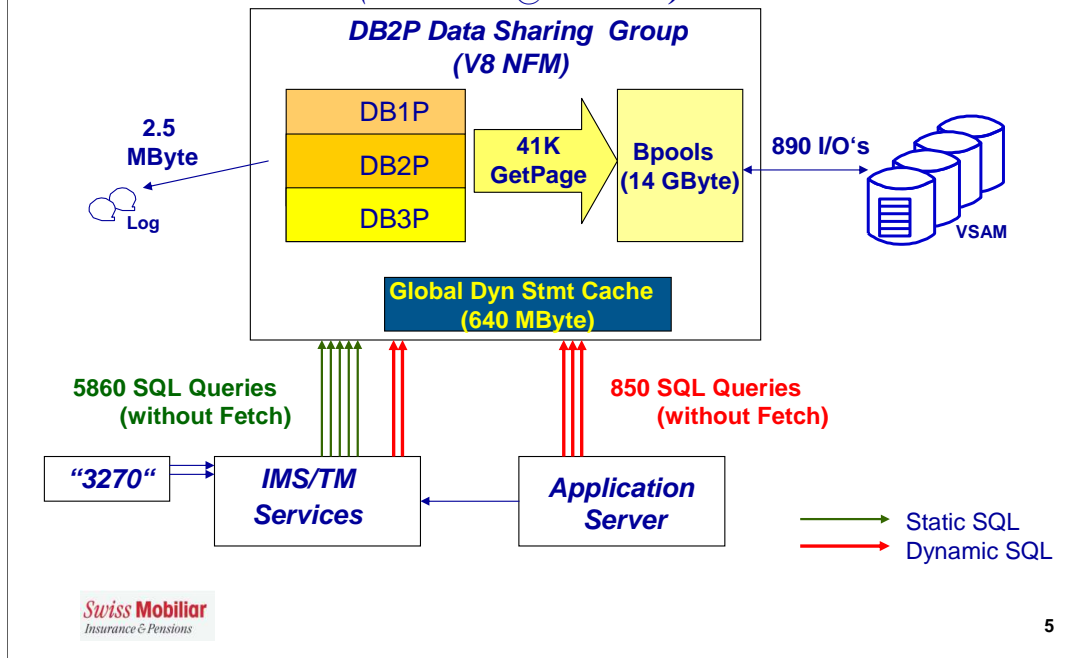
- Swiss Mobiliar Overview
- Dynamic Statement Cache
 - 10 Questions and Answers
- Virtual Bufferpools
 - 10 Questions and Answers
- Group Bufferpools
 - 1 Question and Answer
- Summary (Memory Efficiency Check)

Disclaimer

- The Information contained in this presentation has not been submitted to any formal Swiss Mobiliar or other review and is distributed on an 'as is' basis without any warranty either expressed or implied. The use of this information is the user's responsibility.
- The procedures, results and measurements presented in this paper were run in either the test and development environment or in the production environment at Swiss Mobiliar in Berne, Switzerland. There is no guarantee that the same or similar results will be obtained elsewhere. Users attempting to adapt these procedures and data to their own environments do so at their own risk. All procedures presented have been designed and developed for educational purposes only.

One Second at Swiss Mobiliar

(2007-11-05 @ 10:30 AM)



This slide shows Swiss Mobiliar's query workload within a one-second time interval. The SQL queries are counted without the fetch operations: 1 OPEN, n FETCH, 1 CLOSE is counted as one single query.

Do you care about the Dynamic Statement Cache's hitratio?

- Dynamic Stmt Cache Efficiency Measures@Swiss Mobiliar (Jan 10, 2008, 10:30 AM):
- Size=500MByte (EDMSTMTC=524288)
 - Hitratio: 96%
 - Number of cached statements: 24429
 - No of new entries / second: 7.55
 - How to calculate see notes page
 - Prereq: EXPLAIN STMTCACHE ALL

```
WITH HITRATIO (HITRATIO, ENTRIES, INSERTS, DELTA) AS
(SELECT FLOAT(S1 - S2) / FLOAT(S3), S4, S5, DELTA
FROM
  (SELECT SUM(STAT_EXEC) AS S1, COUNT(*) AS S2
   FROM DSCT*)
  WHERE STAT_EXEC > 1) T1,
  (SELECT SUM(STAT_EXEC) AS S3
   FROM DSCT*)
  WHERE STAT_EXEC > 0) T2,
  (SELECT COUNT(*) AS S4
   FROM DSCT*)
  WHERE STAT_EXEC >= 0) T3,
  (SELECT COUNT(*) AS S5
   FROM DSCT*) WHERE STAT_EXEC >= 1) T4,
  (SELECT DEC(MIDNIGHT_SECONDS(MAX(EXPLAIN_TS))
             - MIDNIGHT_SECONDS(MIN(STAT_TS))) AS DELTA
   FROM DSCT*) WHERE STAT_EXEC>0 ) T6 )

SELECT DEC(HITRATIO, 5, 2) AS HITRATIO
      , ENTRIES AS NUMBER_OF_CACHED_STATEMENTS
      , DEC(FLOAT(INSERTS)/FLOAT(DELTA), 5, 2)
      AS NEW_ENTRIES_PER_SECOND
FROM HITRATIO
```

*)DSCT := DSN_STATEMENT_CACHE_TABLE

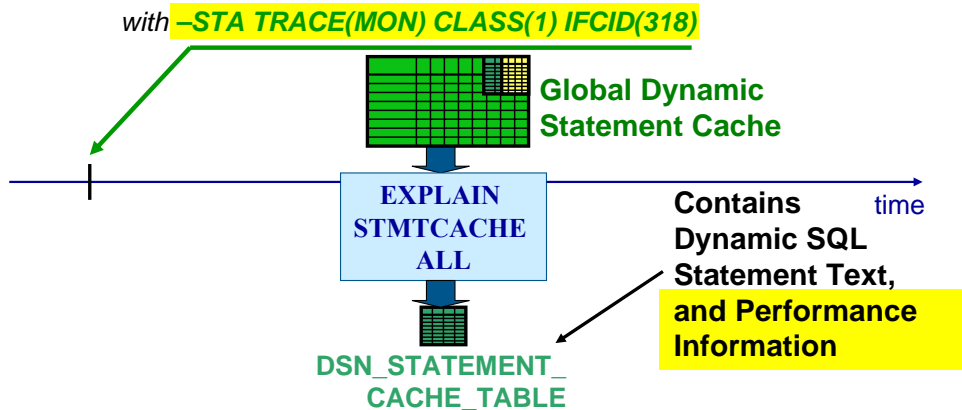
How many statements are stored in your Dynamic Statement Cache?

- Minimum cache residency time
 - 48 min @Jan10,2008,10:30 AM
 - how to calculate see notes page
- No of different access paths
 - 926 @Jan10,2008,10:30 AM
 - Remember: No of cache entries = 25K
- % of cached, but not re-referenced queries (15 min)
 - 67% (13367 queries) @Jan10,2008,10:30 AM

```
SELECT DEC((ENTRIES -
            (ENTRIES*HITRATIO)/LOG(ENTRIES)) /
            (INSERTS/DELTA) / 60) AS MIN_RES_TIME
FROM (SELECT FLOAT(S1 - S2) / FLOAT(S3) AS
      HITRATIO,
      FLOAT(S4) AS ENTRIES, FLOAT(S5) AS INSERTS,
      FLOAT(T) AS DELTA
FROM (SELECT SUM(STAT_EXEC) AS S1,
      COUNT(*) AS S2
      FROM DSCT WHERE STAT_EXEC > 1) T1,
      (SELECT SUM(STAT_EXEC) AS S3
      FROM DSCT WHERE STAT_EXEC > 0) T2,
      (SELECT COUNT(*) AS S4
      FROM DSCT WHERE STAT_EXEC >= 0) T3,
      (SELECT COUNT(*) AS S5
      FROM DSCT WHERE STAT_EXEC >= 1) T4,
      (SELECT MIDNIGHT_SECONDS(MAX(EXPLAIN_TS)) -
      MIDNIGHT_SECONDS(MAX(STAT_TS)) AS T
      FROM DSCT WHERE STAT_EXEC >= 1) T5) T
```

Dynamic SQL Performance Diagnosis

- How do you get meaningful information out of the statement cache? What is a reasonable size?
 - EXPLAIN STMTCACHE ALL



The collection of statistics for statements in the dynamic statement cache can increase the processing cost for those statements. When IFCID 0318 is inactive, DB2 tracks the statements in the dynamic statement cache, but does not accumulate the statistics as those statements are used. When you are not actively monitoring the cache, you should turn off the trace for IFCID 0318.

EXPLAIN STMTCACHE ALL

STAT_EXEC	STAT_CPU	STMT_TEXT
1	0.00112563377501918	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
4	0.00592846889414039	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
3	0.003950138579981	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
23	0.0325548583678171	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
6	0.011235577057974	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
5	0.00960597489394394	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
2	0.00240673673174871	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
1	0.00444850046552864	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
4	0.0339880520514413	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...
10	0.0236521074584886	SELECT C97251_G, C97150_G, C97227_G, C97163_G, C97364_G, C99991_G, C99992_G, C99996_G, C971...

Many statements look quite the same...

There are two questions which arise after inspecting the EXPLAIN STMTCACHE ALL output:

1. Would a larger cache help? Or would it just increase the time for the EXPLAIN STMTCACHE ALL statement and similar operations?
2. What about performance tuning: How could these many entries be avoided? How can you group them together to find out which statements or statement groups are really critical?

Which reports do you use based on EXPLAIN STMTCACHE ALL?

- Key problem to dynamic query tuning: too many queries!
 - identical statement text, but different constants
 - identical statement text, but different authids
 - similar statements, but with same access path
- These statements need to be aggregated, for example:
 - according to statement text, ignoring different values of constants
 - according to the access path used at run time

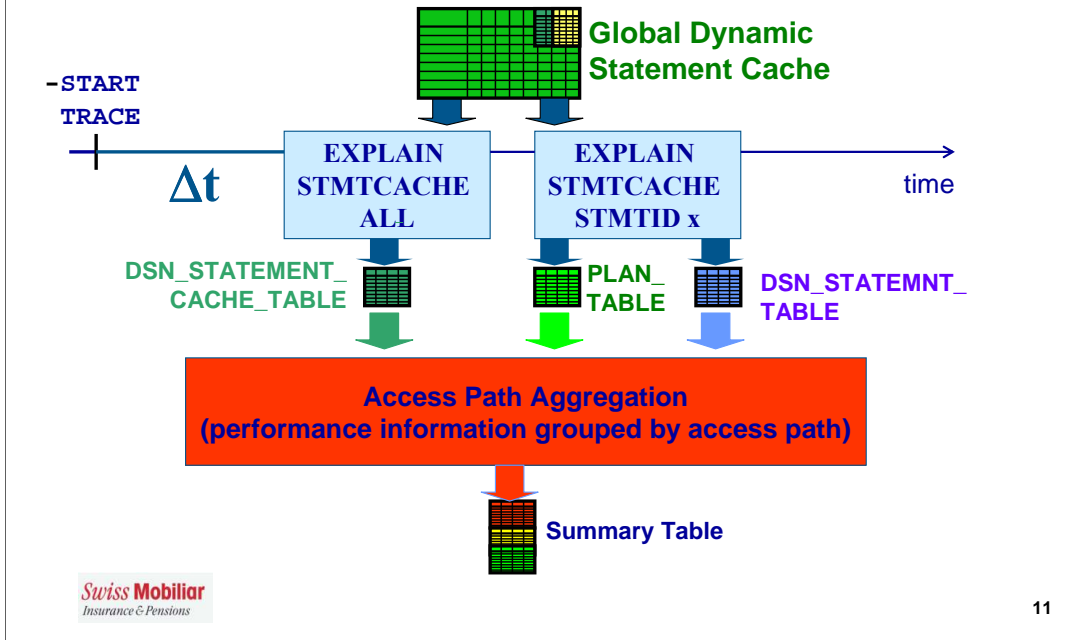
One of the main problems of dynamic SQL tuning is the aggregation of performance information. DB2 for z/OS V8's EXPLAIN STMTCACHE ALL delivers raw data on a very granular level that needs to be aggregated into statement groups.

At Swiss Mobiliar, we group all statements into statement groups according to their access paths used at run time. Thresholds and analysis are applied to these statement groups: This allows the user to quickly identify the access paths which need attention and improvement, without even looking at the query text itself.

We have developed a framework around the EXPLAIN STMTCACHE ALL query, which aggregates the executed queries among their access paths. We named this framework 'M3' according to a powerful car produced and tuned by BMW. M3 guides the user through a comprehensive tuning methodology for evaluating the performance of a database. Key performance metrics are measured, and the users are given a comprehensive analysis of their dynamic SQL workload and a number of tuning ideas.

M3 is available at no cost at Swiss Mobiliar (mail to: thomas.baumann@mobi.ch).

How Do You Aggregate Queries among Access Paths?



The `DSN_STATEMENT_CACHE_TABLE` is also populated with a `STMTID` column, which contains a unique value for each row. Using this value in several `EXPLAIN STMTCACHE STMTID` queries (one for each stmtid), the `PLAN_TABLE` and `DSN_STATEMENT_TABLE` will also be populated – with optimizer information taken directly from the dynamic statement cache. There is no re-evaluation of the access path at this moment, it's just about writing out the stored access path (and cpu consumption preview respectively) information.

Eventually, we join the three tables according to the `stmtid`, and produce performance information group by access path.

What is the Result Out of It?

- M3 Overview Report



M3 Summary Table



Advanced Query Tuning					
M3 V2.2					
				Refresh	
Dynamic SQL Performance Diagnosis for DB2P				Trace start 2007/9/24 10:11 am Cache snapshot 2007/9/24 10:27 am	
Emergency Tuning					
ET01	No of Stmt Groups with high CPU / Statement Ratio	0	0%	0	0%
ET02	% of Unreferenced Cached Statements	56%	-5%	66%	0%
Performance Control: SQL Statement					
PC01	No of Stmt Groups with Inefficient Predicates	0	0%	0	0%
PC02	No of Stmt Groups with Resource Intensive Sorts	0	0%	0	0%
PC03	No of Stmt Groups with Intensive Locking	0	-100%	0	-100%
PC04	No of Stmt Groups with Optimizer Challenges	0	0%	0	0%
Performance Control: Memory Management					
PC05	Dyn Statement Cache Min Residency Time (min)	44	-10%		
PC06	Local Buffer Pools Min Residency Time (min)	23	-12%		
PC07	No of Failed Writes to Group Buffer Pool	0	0%		
PC08	No of Sync I/Os caused by Cross-Invalidation (per min)	0	0%		
Performance Control: Data Management					
PC09	Getpage requests / second	2264	11%	2091	4%
Performance Management					
PM01	Tspce/Index Runstats Efficiency	n/a	n/a	n/a	n/a
PM02	Tspce/Index Reorg Efficiency	n/a	n/a	n/a	n/a
PM03	Optimization Potential	62%	3%	57%	2%
PM04	Unused indexes	20	11%	n/a	n/a
PM05	Total CPU Time per second	0.08500	4%	0.07147	-7%
PM06	Total Sync I/O Time per second	1.59921	11%	1.38093	8%
PM07	Total Elapsed Time per second	1.94253	11%	1.59572	7%
PM08	CPU Seconds / Processed Row	0.00004	-50%	0.00007	-13%
PM09	Elapsed Seconds / Processed Row	0.00096	-45%	0.00166	3%
DynSQL Performance Index (100=31.08.2007) DSPI		206.19	92.59%	118.32	7.52%

12

If your organization manages dozens of applications and hundreds of databases, wouldn't you like to be able to quickly determine if your dynamic SQL statements require your team's attention?

If there is a poor performance or inefficiency problem, M3 will find it and bring it to your attention. M3 guides the user through a comprehensive tuning methodology for evaluating the performance of a database. Key performance metrics are measured, and the users are given a comprehensive analysis of their dynamic SQL workload and a number of tuning ideas.

The overview report represents the latest time interval's measurements, compared with the previous one. It quickly identifies tuning opportunities, and highlights problem regions which require further attention.

A Dynamic SQL Performance Score, like a Credit Score, is also provided so that you can quickly understand your workload's health and efficiency. You no longer need to digest dozens of metrics and indicators.

Detail reports provide further details to the overview report. They can be executed by any SQL query processor and need no further adjustment.

Two sample detail reports will be further discussed on the next two pages.

Which queries qualify for ,inefficient search attributes?‘

$$\frac{\text{Number of rows examined}}{\text{Number of rows fetched}} > 10$$

$$\& \sum_h \text{CPU per Statement Group}^*) > 30 \text{ sec}$$

*) a *Statement Group* consists of all SQL queries using the same access path

This report selects all access paths which consume more than 30 seconds CPU per hour, and which examine (within Data Manager or RDS) more than 10 rows for each row fetched. In other words: More than 10 rows must be checked in order to find one qualifying row.

This could be because of missing indexes, stage-2-predicates, inaccurate catalog statistics or other reasons.

Which queries challenge the optimizer?

$$\frac{\text{measured CPU (dsn_statement_cache_table)}}{\text{estimated CPU (dsn_statemnt_table)}} > 5$$
$$\& \sum_h \text{CPU per Statement Group}^*) > 20 \text{ sec}$$

*) a *Statement Group* consists of all SQL queries using the same access path

The following query selects statements which qualify for 'optimizer challenging queries', which means that the measured CPU time after execution is at least 5 times higher than the predicted CPU consumption. Queries that qualify for this category should be checked for accurate catalog statistics. Often, skewed data distributions lead to such results, and additional RUNSTATS options might be required.

```
SELECT STMT_TEXT
FROM DSN_STATEMENT_CACHE_TABLE C
JOIN DSN_STATEMNT_TABLE S
ON C.STMT_ID = S.QUERYNO
AND C.COLLID = S.COLLID
WHERE
(1000*C.STAT_CPU / C.STAT_EXEC)
/
S.PROCMS > 5
```

Which queries challenge the optimizer? Example.

PROGRAM_NAME	CURSQLID	STAT_CPU	NO_OF_ENTRIES	NO_OF_EXECUTIONS	SAMPLE_QUERY	APIID
SYSLN300	ORPTOOL	3.06096009533256	1	9	SELECT DISTINCT CASE WHEN DB2P...	DB2PROD.TORPRGARONS1DB2PROD:XORP...

```
SELECT DISTINCT CASE WHEN DB2PVIEW.VORPART1.C39992 >= current timestamp AND DB2PVIEW.VORPART1.C39991 <= current timestamp THEN '0' ELSE '1' END AS orderflag, DB2PVIEW.VORPART1.C43709 AS C43709,
DB2PVIEW.VORPART1.C63570 AS C63570, DB2PVIEW.VORPART1.C63572 AS C63572, DB2PVIEW.VORPART1.C63574 AS C63574, DB2PVIEW.VORPART1.C63578 AS C63578, DB2PVIEW.VORPART1.C63588 AS C63588,
DB2PVIEW.VORPART1.C63592 AS C63592, DB2PVIEW.VORPART1.C63593 AS C63593, DB2PVIEW.VORPART1.C43985 AS C43985, DB2PVIEW.VORPART1.C39712 AS C39712, DB2PVIEW.VORPART1.C63590 AS C63590,
DB2PVIEW.VORPART1.C63597 AS C63597, DB2PVIEW.VORPART1.C44014 AS C44014, DB2PVIEW.VORPART1.C39930 AS C39930, char(DB2PVIEW.VORPART1.C39991) AS C39991, char(DB2PVIEW.VORPART1.C39993) AS C39993,
char(DB2PVIEW.VORPART1.C39993) AS C39993, DB2PVIEW.VORPART1.C39994 AS C39994, char(DB2PVIEW.VORPART1.C39995) AS C39995, DB2PVIEW.VORPART1.C39996 AS C39996 FROM DB2PVIEW.VORPART1 LEFT OUTER JOIN
DB2PVIEW.VORPART1 ON (DB2PVIEW.VORPART1.C43709 = DB2PVIEW.VORPART1.C43709 AND DB2PVIEW.VORPART1.C39991 <= DB2PVIEW.VORPART1.C39992 AND DB2PVIEW.VORPART1.C39992 >= DB2PVIEW.VORPART1.C39991 AND
DB2PVIEW.VORPART1.C39993 < DB2PVIEW.VORPART1.C39995 AND DB2PVIEW.VORPART1.C39995 = DB2PVIEW.VORPART1.C39993) INNER JOIN DB2PVIEW.VORPART1 ON (DB2PVIEW.VORPART1.C43709 = DB2PVIEW.VORPART1.C43709
AND DB2PVIEW.VORPART1.C39991 <= DB2PVIEW.VORPART1.C39992 AND DB2PVIEW.VORPART1.C39992 >= DB2PVIEW.VORPART1.C39991 AND DB2PVIEW.VORPART1.C39993 < DB2PVIEW.VORPART1.C39995 AND
DB2PVIEW.VORPART1.C39995 > DB2PVIEW.VORPART1.C39993) WHERE DB2PVIEW.VORPART1.C39996 = ? ORDER BY C43709, C39995 DESC, orderflag, C39992 DESC
```

Do you really want to care about the **query**?
Or do you prefer to analyze the **access path**?

```
DB2PROD.TORPRGA R
NLJ DB2PROD.XORPART1 I2 N
NLJ DB2PROD.XORPADR1 I1 N
SORT
```

In this example we detected a tablespace scan for table DB2PROD.TORPRGA, which consumed more CPU than predicted (probably because the prediction was done against an empty tablespace). Freshly updated catalog statistics (by the RUNSTATS utility) would probably let this query disappear from this specific report, but it would shortly reappear in the 'inefficient predicate' report, as long as there is no index created and applied for this query.

How can you identify unused indexes? (V8)

- Unused Indexes
 - Does not exist in SYSPACKDEP.BQUALIFIER/BNAME
 - Does not exist in PLAN_TABLE after EXPLAIN STMTCACHE STMTID
- V9 Enhancement
 - SYSIBM.SYSINDEXSPACESTATS.LASTUSED

Querying the PLAN_TABLE after EXPLAIN STMTCACHE STMTID for all stmtids reported in EXPLAIN STMTCACHE ALL, and comparing this information to SYSIBM.SYSPACKDEP to identify indexes used in statically bound packages displays all indexes never used for read (select statements or subquery of insert, update or delete statements) operations. As per default, we consider the top 20 tablespaces regarding update frequency only. Optionally, additional tablespaces might be included in the report's scope.

Which are your most important access paths?

Top 5 access paths regarding CPU usage

TOTAL_CPU	NO_OF_ENTRIES	NO_OF_EXECUTIONS	SAMPLE_QUERY	APIID
3.887526	42	8723	Insert into DB2PVIEW.VP..	0DB2PRODTPA97920N
3.298127	1	12458	Insert into DB2PVIEW.VE..	0DB2PRODTEQ_TPA09410N
3.060960	1	9	SELECT DISTINCT CAS..	0DB2PRODTORPRGARONS1DB2
2.899993	1	11479	Insert into DB2PVIEW.VP..	0DB2PRODTPA09410N
2.834555	1	11570	Insert into DB2PVIEW.VE..	0DB2PRODTEQ_TPA97920N

Top 5 access paths regarding elapsed time

TOTAL_ELAPSED	NO_OF_ENTRIES	NO_OF_EXECUTIONS	SAMPLE_QUERY	APIID	SYNCDIO_PER_EXEC	GETPAGE_PER_EXEC
110.522114	365	1464	SELECT C97251_G, C97150_G, C9..	0DB2PVIEWMV714601R0..	5	54
108.838947	42	8723	Insert into DB2PVIEW.VPA97922 (C9..	0DB2PRODTPA97920N	0	9
103.938526	1	11479	Insert into DB2PVIEW.VPA09412 (C9..	0DB2PRODTPA09410N	0	8
79.227983	26	39	SELECT DB2PVIEW.VSAMELD1.C4..	0DB2PRODXSAMELD211N..	1	1449
68.827823	1	3066	Insert into DB2PVIEW.VPA0800A (C6..	0DB2PRODTPA08000N	1	15

These are the top reports (the first we look at) from our dynamic statement cache analysis process. The top access paths regarding CPU consumption (or, for the second report, elapsed time) are presented. This gives an excellent overview of our workload and helps to quickly understanding which objects are most frequently accessed, and by which access paths they are accessed.

Many performance tuning actions can be derived of these two reports!

64 bit Buffer Pools

- *With 64bit address spaces and large bufferpools, is it still necessary to care about object placement, or should I rather define one single very large bufferpool with all objects in it?*
 - Efficiency Measures
 - Bufferpool Strategy
 - What about 2GByte Extra Memory?

„One Size doesn‘t fit all“ has long been a widely accepted paradigm for bufferpool tuning. Should I change this approach, now that very large bufferpools are possible?

This is my personal opinion:

- Use different bufferpools only if the characteristics of the pools are different (for example, different values for page stealing method)
- Display commands are sufficient for bufferpool tuning, object placement based on IFCID 198 performance traces (getpage information) is not necessary any longer.
- There are always some exceptions which justify additional, specific pools for specific objects.

The following slides will further explain this opinion, and eventually, a new approach for bufferpool tuning will be presented: The 2GByte Impact approach.

What is a large Buffer Pool?

- Before V8, 100,000 pages was a large pool
 - 400 MByte (4K Pages)
- With V8,
 - 100,000 pages (400MByte) is medium size
 - 1,000,000 pages (4GByte for 4K Pages) **and above** is a large pool

64bit architecture allows the definition of much larger memory resources, such that the common perception of a large bufferpool has changed by a factor of 10. What has been large is now small to medium, and a new world of really large pools is there to be used.

Efficiency Measures (1/3)

- *For many years I was told that the hitratio is the most important value, but recently I heard about other metrics such as sync I/O rate or page residency time.*

- **Hitratio Advantages**

- Simple to Calculate from DIS BPOOL command:

$\frac{\text{Random Getpage} - \text{Sync Read I/O (R)}}{\text{Random Getpage}}$
--

- Useful to compare same pool with same workload across time

- **Hitratio Disadvantage**

- Cannot be used to compare between pools

The formula described is also known as *'random hitratio'*. There is also an *overall hitratio* and a *prefetch hitratio* available, but both of them are of fewer interest.

The main disadvantage of the hitratio as an efficiency metrics is the hitratio's dependency of the current workload. Just imagine the following query:

```
SELECT * FROM CUSTOMER_ACCOUNT A JOIN CURRENCY_DESCRIPTION C
ON    A.CODE = C.CODE
WHERE A.CUSTOMER_NO BETWEEN ? AND ?
```

This query will scan through an important part of the (large) customer_account table, and with every row examined, there will be a random getpage for an index to the currency table. It is highly probable that all these accesses to the currency table will result in bufferpool hits, independent of the size and configuration of the currency tablespace's bufferpool. Therefore, the hitratio is massively impacted by this one single query.

Comparison of hitratios between different time frames are meaningful only if the workload is identical, and comparisons of the hitratio between different bufferpools do make no sense.

Efficiency Measures (2/3)

- *What is a reasonable minimum page residency time, and how can I measure it?*
- **Minimum Page Residency Time Advantages**
 - Simple to Calculate from DIS BPOOL command*):

$$\frac{\text{Buffers active} * (1 - \text{VPSEQT}/100)}{\text{Sync Read I/O (R)}}$$

- Useful to compare between bufferpools
- Useful to compare same pool over time

*) see notes page for more detailed formula

The formula presented above is correct for bufferpools using the FIFO page stealing method only.

As with hitratio, there is an *overall* residency time, a *random* residency time, and a *prefetch (sequential)* residency time available. We only consider the *random* residency time as of interest.

Sequential I/O for large bufferpools should not be a problem (as long as SYNC READ I/O (S) is very small compared to SYNC READ I/O (R)). Both values can be found using the -DIS BPOOL (bpname) DETAIL command. Remember that in DB2 V9, the sequential page read quantity for 4K pools will be extended from 32 pages to 64 pages for medium and large bufferpools (if VPSEQT*no of buffers > 40000).

The formula presented in the slide above is correct for FIFO buffer pools only. The approximation formula (based on queuing theory) for LRU-ruled pools includes chain management, and results in shorter residency times:

$$\frac{\#pages - (\#pages * \text{hitratio}) / \log(\#pages)}{\text{No of sync I/O per second}}$$

(#pages := VPSIZE * (1 - VPSEQT/100))

Efficiency Measures (3/3)

- *Do you use other efficiency measures?*
 - Page Re-read Count
 - No of Sync I/Os for pages re-read read into the pool in a given interval of time (*avoidable reads*)
 - Needs IFCID 198 performance trace
 - For today's large bufferpools not useful anymore
 - **Sync I/O rate**
 - No of Sync I/O per second
 - „2GByte Impact“
 - Sync I/O rate (@VPSIZE+2GB) – Sync I/O rate

The final goal of bufferpool memory tuning is the reduction of the sync I/O rate. This value is the most important one, all other only do help to analyze the bufferpool configuration.

Bpool Tuning Key Questions

- How many pools? Which object into which pool?
- What values should be used for
 - VPSIZE
 - PGFIX: YES or NO?
 - PGSTEAL: LRU or FIFO?
 - VPSEQT (and VPPSEQT)
 - DWQT (and VDWQT)
- Basic Rule: Different bufferpools only if these values are different!
- Use `-DIS BPOOL DETAIL` to monitor values

The PGFIX option is useful where the I/O rate is high. It fixes buffers in real storage to avoid page fix/free for each I/O operation.

Real storage must be available to back the pool (DSNB541I message will be issued and PGFIX will be ignored if the total number of active buffers is larger than 80% of real storage).

PGSTEAL method FIFO is useful where the I/O rate is very low; it avoids unnecessary chain management for pages pinned permanently to the buffer pool.

The combination of these two options might save up to 10% of CPU.

What is your personal Bufferpool Strategy (1/2)?

- BP0: Catalog+Directory
- BP1: All regular tablespaces
- BP2: All regular indexes
- BP3: Highly Frequented Small Objects (ReadOnly)
- BP4: Extremely frequently updated small objects
- BP5: Problem Investigation
- BP6: Workfiles
- BP7: High Priority Applications

Random vs. Sequential solved by VPSEQT

BP0 should be used exclusively by catalog and directory objects, basically to avoid any availability problems, especially in a data sharing environment. Residency Time should be high.

For BP1 and BP2, consider PGFIX YES, as these pools will have a high I/O rate, and reduce DWQT and VDWQT to enforce trickle writes.

Use FIFO for BP3 in order to avoid chain management and to avoid latch contention. VPSEQT could be set to 0 if Sync Read I/O rate is 0 to avoid scheduling of prefetch I/O.

Size BP6 large enough to ensure that small sorts are performed in memory, and set VPSEQT=100. Set VDWQT and DWQT much higher than default in order to avoid unnecessary writes.

BP4 is similarly parametrized as BP6, and BP7 like BP1/BP2.

Heavy sequential processing could be routed to a specific data sharing member with smaller bufferpools.

What is your personal Bufferpool Strategy (2/2)?

BP	Data type	VPSIZE	VPSEQT	DWT	VDWT	Method	PGFix
0	Catalog	10K	80%	50%	10%	LRU	NO
8K0		25K	80%	50%	10%	LRU	NO
16K0		2K	80%	50%	10%	LRU	NO
1	Data	1.0M	20%	10%	5%	LRU	YES
2	Index	1.25M	20%	50%	10%	LRU	YES
3	In-memory Heavy-read	40K	0%	50%	10%	FIFO	NO
4+ 7	In-memory Heavy-upd	725K	0%	90%	90%	FIFO	NO
5	Pipeline	5K	80%	80%	50%	FIFO	NO
6	Workfile	20K	100%	70%	50%	LRU	NO

Swiss Mobiliar
Insurance & Pensions

25

This slide shows Swiss Mobiliar's bufferpool definitions as of Nov 5, 2007 (the day of the one-second picture at page 5).

The big picture has been remained the same since that point in time, but minor changes took place (see next pages for an example of such changes).

Sample Bufferpool Sizing: Trial and Error

- Use feedback from
 - hitratio, residency time, sync I/O rate

Name	VPSIZE	Getpage rate	Hitratio	Residency Time	Sync I/O rate
BP0	10K	1822	100%	permanent	0
BP1	1M	3343	95.3%	49 min	156
BP2	1.25 M	21172	98.0%	23 min	417
BP3	40K	2582	100%	permanent	0
BP4	40K	1716	99.9%	300 min	1
BP7	685 K	1475	98.2%	338 min	26

Based on hitratio only, we would increase BP1, and decrease BP2, given the same amount of total memory available.

Our final goal is to reduce the sync I/O rate (at a given getpage rate and running the same workload).

Therefore, our analysis starts with the residency time:

49 min for BP1 looks to large, compared to 23 min for the index buffer pool.

What about an increase of BP2 and a decrease of BP1?

Sample Bufferpool Sizing: Trial and Error

- Move 2GB Memory from BP1 to BP2:

Name	VPSIZE	Getpage rate	Hitratio	Residency Time	Sync I/O rate
BP1	1M → 0.5M	3343 → 2641	95.3% → 93.7%	49 min → 24 min	156 → 164
BP2	1.25 M → 1.75M	21172 → 19692	98.0% → 98.1%	23 min → 37 min	417 → 360

- ‚Overall Hitratio‘ BP1+BP2: **97.67%** → **97.65%**

Maybe this was too much. The *overall hitratio*^{*)} of BP1+BP2 shows a little decrease, the total number of sync I/O's is a little bit higher (if calculated on the same number of getpages).

As these changes and measurements were done within a short period of time, we were quite sure that the workload behind these numbers was very similar.

*) overall hitratio (BP1+BP2) :=

$$\frac{\#Getpg_{BP1} + \#Getpg_{BP2} - \#SyncI/O_{BP1} - \#SyncI/O_{BP2}}{\#Getpg_{BP1} + \#Getpg_{BP2}}$$

Sample Bufferpool Sizing: Trial and Error

- Move 1GB Memory back from BP2 to BP1:

Name	VPSIZE	Getpage rate	Hitratio	Residency Time	Sync I/O rate
BP1	1M	3343	95.3%	49 min	156
	→0.5M	→ 2641	→93.7%	→24 min	→ 164
	→0.75M	→ 3327	→94.6%	→ 29 min	→ 179
BP2	1.25M	21172	98.0%	23 min	417
	→1.75M	→19692	→98.1%	→37 min	→ 360
	→1.5M	→21553	→98.2%	→ 25 min	→ 378

- ,Overall Hitratio' BP1+BP2: **97.67%** → **97.65%** → **97.76%**

This looks better; the residency time of both the tablespace and the index buffer pool are very similar now.

Can DB2 manage automatically the bufferpool size?

- DB2 9: AUTOSIZE=YES
 - Integrated with WLM
 - Increase/decrease up to 25% of initial size
 - Based on random hitratio
 - Is hitratio useful to compare different pools?

This slide is derived from the manuals, not from practical experience. Therefore, I can not make any further comment to it.

What benefit could I expect from 2GByte additional memory for my buffer pools?

- “2GByte Impact”
- Prerequisite: 2 GByte extra memory
 - reduce VPSIZE of BP1 and BP2 (or BP7) first
- How it Works
 - Add 2GByte and compare Sync I/O rate
 - Useful for BP1 and BP2 in our configuration
 - If Sync I/O will be reduced by >20%, buy 2GByte more memory, and repeat scenario
 - Otherwise, redistribute to BP1/BP2(/BP7)

The “2 Gbyte Impact“ is a new metrics applicable to large bufferpools. The basic idea is as follows:

As long as the addition of 2 Gbyte extra memory (500,000 pages for a 4K pool) results in a reduction of more than 20% of the sync I/O rate, it is worth investing in memory!

To be able to increase VPSIZE by 2 Gbyte, you must first reduce this amount by reducing the VPSIZE of other pools. With this approach, the latest 2GByte-portion will decrease the sync I/O rate by less than 20% and will therefore made undone, leaving 2GByte of memory to be used otherwise (I have a constant ,overallocation*‘ of 200,000 pages in BP1 and BP2, and 100,000 pages in BP7).

Before any new ,2GByte impact measurement‘, I reduce BP1/BP2/BP7 by this size, start the scenario, and redistribute the memory back afterwards. This is done after major application changes, 2 or 3 times per year.

*‘ compared to the bufferpool tuning goals. All of these pool‘’s resources must be backed by real storage.

What's currently in your bufferpools?

- -DIS BPOOL LSTATS SP(..)
 - Allows to monitor BP and I/O activity at dataset level
 - Incremental counts since last command

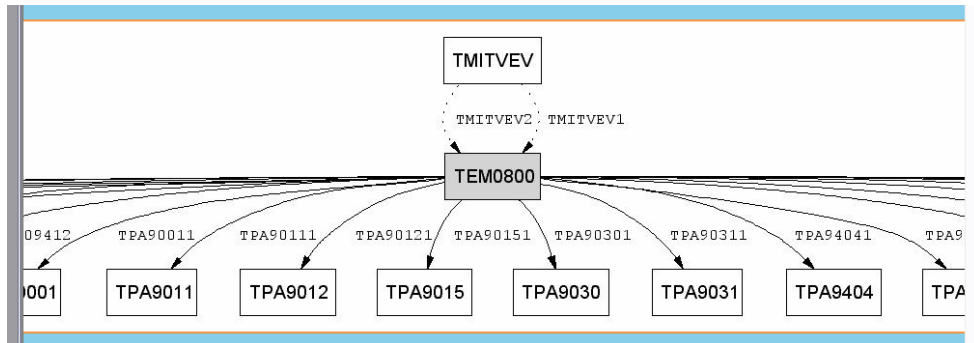
```
DSNB467I  -DB2P  STATISTICS FOR INDEX SPACE DPVERT  .XPA09411 -
           DATA SET #:  20 USE COUNT:      1
DSNB453I  -DB2P  VP CACHED PAGES -
           CURRENT      = 1778  MAX          = 2227
           CHANGED     =    0  MAX          =   36
DSNB455I  -DB2P  SYNCHRONOUS I/O DELAYS -,
           AVERAGE DELAY =    9  MAXIMUM DELAY =  554
           TOTAL PAGES  = 1336
DSNB456I  -DB2P  ASYNCHRONOUS I/O DELAYS -,
           AVERAGE DELAY =    1  MAXIMUM DELAY =    9
           TOTAL PAGES  = 4736  TOTAL I/O COUNT =  236
```

The -DIS BPOOL LSTATS command might be a useful tool both for bufferpool tuning, as for problem investigation (troubleshooting tuning).

Two examples will follow.

DIS BPOOL – Example 1

- Which parent-child relation is responsible for long response time if you delete the parent key?



There are more than 50 dependent tables (referential integrity ,children‘) of parent table TEM0800. If we delete a key in TEM0800, all referential integrity constraints need to be checked. Usually, this is performed by an index access to the dependent table. But only if an index which corresponds to the columns of the referential integrity constraint exists in the dependent table.

To identify which of the dependent tables will be scanned by a tablespace scan instead of an index access, we refer to a `-DIS BPOOL LSTATS` command:

```
-DIS BPOOL(BP1)LSTATS SP(SY*)
```

All the candidate tablespaces have SY* as tablespace name prefix. The command output will be presented on the next slide.

DIS BPOOL – Example 1

```
DSNB467I -DB2Y STATISTICS FOR TABLE SPACE DYVERT .SYBA0800 -
          DATA SET #: 2 USE COUNT: 1
DSNB453I -DB2Y VP CACHED PAGES -
          CURRENT      = 8 MAX          = 1823
          CHANGED     = 8 MAX          = 8
DSNB455I -DB2Y SYNCHRONOUS I/O DELAYS -
          AVERAGE DELAY = 1 MAXIMUM DELAY = 1
          TOTAL PAGES  = 2
DSNB456I -DB2Y ASYNCHRONOUS I/O DELAYS -
          AVERAGE DELAY = 0 MAXIMUM DELAY = 0
          TOTAL PAGES  = 3628 TOTAL I/O COUNT = 114

DSNB467I -DB2Y STATISTICS FOR TABLE SPACE DYVERT .SYPA9001 -
          DATA SET #: 1 USE COUNT: 1
DSNB453I -DB2Y VP CACHED PAGES -
          CURRENT      = 1 MAX          = 63
          CHANGED     = 0 MAX          = 0
DSNB455I -DB2Y SYNCHRONOUS I/O DELAYS -
          AVERAGE DELAY = 0 MAXIMUM DELAY = 0
          TOTAL PAGES  = 2
```

There is only one tablespace with very many *asynchronous* I/O operations, and therefore it is easy to identify the tablespace which is responsible for the very long response time of the delete query.

DIS BPOOL – Example 2

- *My favourite user started a long running SQL statement 30 minutes ago. How can I figure out how long it will take until it ends? Do you know any SQL statement progress monitor?*
 - Query Processing Time Preview:
 - Identify sequentially accessed object (by inspecting access path): outer table of a join
 - Use –DIS BPOOL LSTATS SP(objectname) to count number and frequency of asynchronously read pages
 - Compare with total number of pages

With the same technique, it is possible to analyze the query progress of a long running sql statement: Just count the number of sequentially accessed pages, and compare this number with the total number of pages of this object.

Which metrics should I really take care of for my Group Buffer Pools?

- Group Bufferpool Efficiency Measures
 - Avoidable Sync I/Os
 - Failed Writes

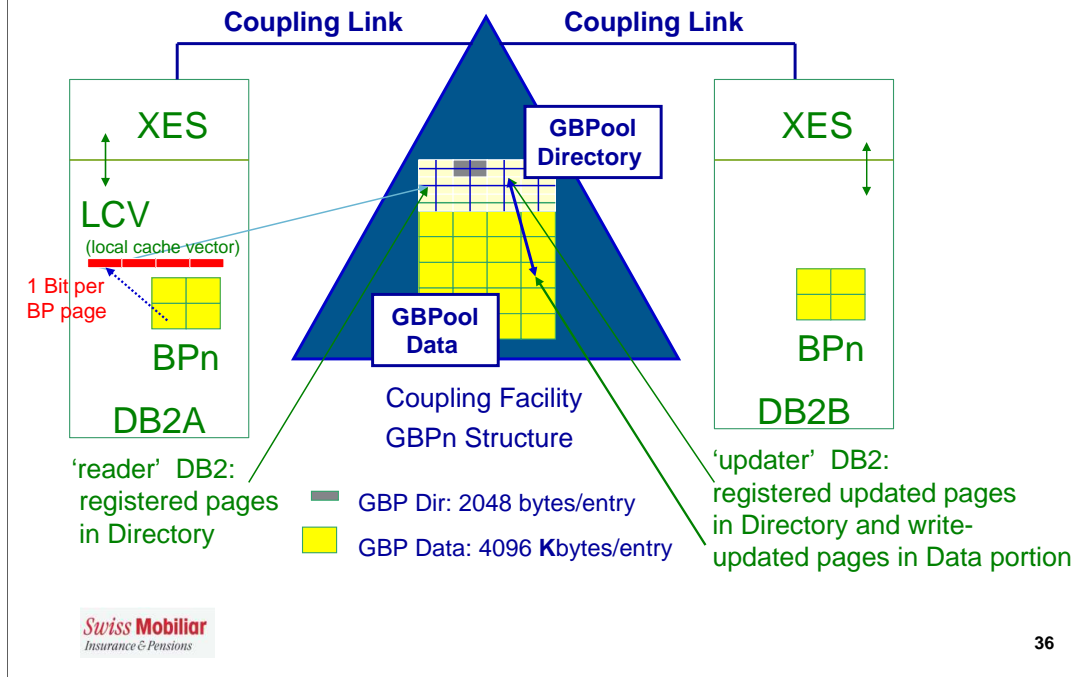
In contrast to the virtual buffer pools, I do not look at the group buffer pools as a means for query tuning. But these structures and their parameters should nevertheless be defined large enough and require some attention to avoid query performance problems.

In other words: Group Bufferpools might cause performance problems (if defined too small), but there is no additional query performance benefit if they are defined larger than necessary.

The following slides will discuss two topics which should be periodically checked (avoidable synchronous I/Os due to cross invalidation caused by directory reclaims, and failed writes due to unavailable storage).

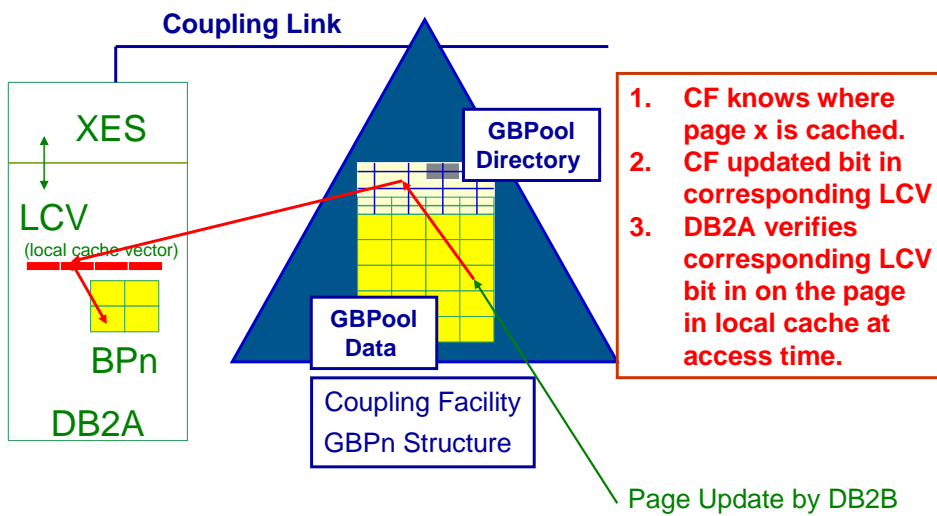
Again, these checks might be done by simple display commands.

Group Buffer Pool Overview



Group Bufferpools are used to make sure that all data sharing members get the most current state of a page. A positive side effect of this feature is the ability to avoid I/Os, which might already have been done by another data sharing member for the same page.

Cross Invalidation



There are two events which lead to ,cross invalidation‘:

1. A page update in data sharing member DB2B cross invalidates the local buffer pool entry for the same page in DB2A.
2. If, during the Page Registration Process, no free slot in the directory is found, the oldest (LRU) directory entry will be deleted and the corresponding page in the local bufferpool will be invalidated (via the same mechanism used during the page-update process). If this page is later re-referenced in the local bufferpool, it needs an additional I/O from DASD.

Group Buffer Pool Directory

- Avoidable Sync I/Os
 - -DIS GBPOOL (GBP_x) MDETAIL

Usually due to writes
(these are ok)

```
DSNB771I  -DB2P INCREMENTAL MEMBER DETAIL STATISTICS
          SINCE 10:11:41, JAN  9, 2008,
DSNB773I  -DB2P MEMBER DETAIL STATISTICS
          SYNCHRONOUS READS
          DUE TO BUFFER INVALIDATION
          DATA RETURNED                = 19
          DATA NOT RETURNED           = 1
```

Usually due to directory reclaims
(avoidable with larger directory)

If a -DIS GBPOOL(GBP_x) GDETAIL command shows many cross invalidations due to directory reclaims, this situation might lead to additional (avoidable) synchronous I/Os if these pages will be re-accessed.

If they really will be re-accessed can be answered by a

-DIS GBPOOL (GBP_x) MDETAIL

command (see output above). This commands represents the member's view to the group buffer pool and must be executed for each member individually.

MDETAIL(*) reports all data counted since group buffer pool allocation, MDETAIL (or MDETAIL(INT)) counts the interval since the last command.

In order to calculate the 'avoidable sync I/O per second' metrics, we use the MDETAIL(INT) option.

Group Buffer Pool Data

- Failed Writes
 - -DIS GBPOOL (GBPx) GDETAIL(*)

```
DSNB786I  -DB2P  WRITES,
           CHANGED PAGES                = 18087748
           CLEAN PAGES                   = 0
           FAILED DUE TO LACK OF STORAGE = 0
           CHANGED PAGES SNAPSHOT VALUE = 97
```

Anything above 0 requires larger data part

For this command we prefer to use the GDETAIL(*) option instead of GDETAIL(INT), because we are interested in the total number of failed writes since group bufferpool allocation time.

These are the key values which we are happy to meet. Specifically for dynamic SQL workload for DB2 for z/OS V8, look at these values for GBP8K0 and GBP16K0, since these buffer pools contain now catalog data necessary for bind (and prepare) operations. If you see writes failed due to lack of storage in these pools, make them higher (10K should be ok), and recalculate elapsed times for bind and prepare operations of complex queries.

Summary (1/2): An Overall Memory Efficiency Check

- **GBPool**
 - ✓ No of failed writes = 0
 - ✓ Sync Reads with Data Not Returned per second < 0.1% of Sync I/O Rate
- **Virtual Pools**
 - ✓ Total Sync I/O rate @ current total VPSIZE > 0.8 * Total Sync I/O rate @ (VPSIZE+2GByte)
- **Dynamic Statement Cache**
 - ✓ No of new cache entries/sec @ current size > 0.8 * No of new cache entries/sec @ size + 50MByte
- **Combine information**
 - Access path information and DIS BP() LSTATS SP(x)

This is the summary of our memory management initiative:

Check if the GBPools are large enough to avoid failed writes and minimize avoidable synchronous I/Os.

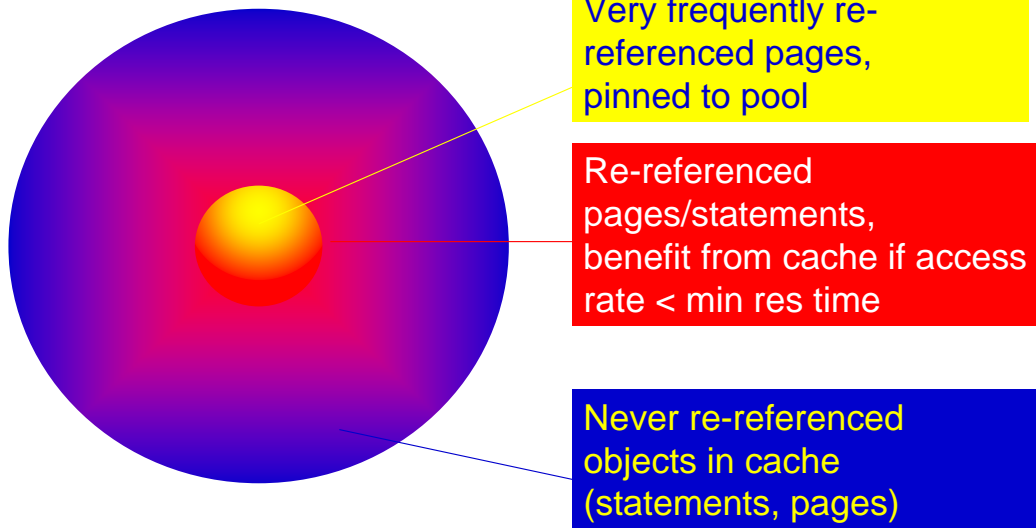
Check if additional 2GBytes of memory would decrease the sync I/O rate by more than 20%.

Check if additional 50 Mbyte of memory would decrease the number of new cache entries into the dynamic statement cache by more than 20%.

Use access path information from dynamic statement cache analysis to identify objects (candidates) for pinning into a bufferpool. Analyze them by using the

-DIS BPOOL(..) LSTATS SP(x) command.

Summary (2/2)



This is the art of memory tuning: Minimize I/O rate while keeping the blue part as small as possible !