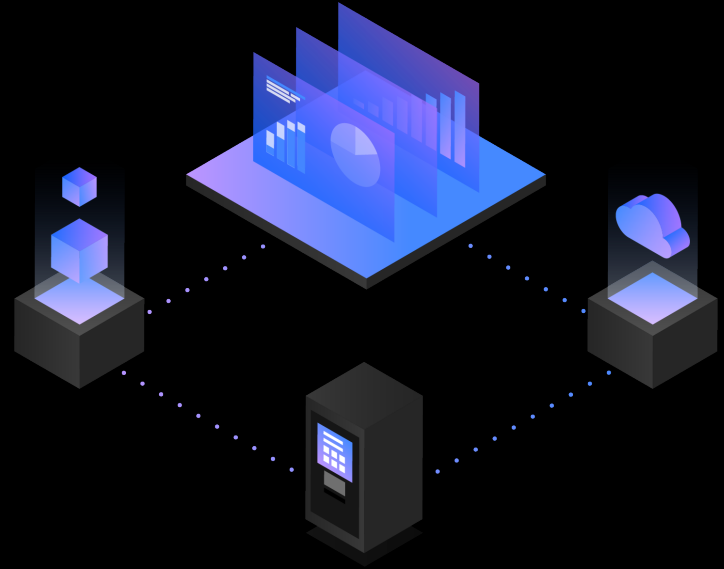


# New Frontiers in Leveraging z/OS Real Storage for Optimal Db2 Performance

Tridex

September 23, 2021

Robert Catterall, IBM  
Senior Consulting Db2 for z/OS Specialist



# Introduction

- The mainframe memory scene, in a Db2 for z/OS context, is **different than it was** just a few years ago
  - z/OS LPAR real storage sizes are getting to be **much larger** – over 100 GB of real storage is not unusual, several hundred GB is increasingly common
  - Db2 for z/OS keeps giving us **new ways to leverage z/OS real storage** for improved application performance and CPU efficiency
    - Recent versions of Db2 have introduced buffer pools backed by large real storage page frames, contiguous buffer pools, high-performance DBATs, a much bigger sort tree, greater use of sparse indexes, and more
- My aim: help you use mainframe memory, and memory-leveraging Db2 features and functions, **as effectively as possible**

# Agenda

- Assessing the real storage situation for a z/OS LPAR
- Buffer pools – your single biggest memory-related leverage point
- Other ways to exploit real storage to enhance Db2 performance

# **Assessing the real storage situation for a z/OS LPAR**

# How much memory should a z/OS LPAR have?

- If it is an LPAR in which a **production Db2** subsystem runs, I would say that it should have ***at least 20 GB of real storage per engine***
  - And by “per engine,” I’m talking about **general-purpose and zIIP engines**
- This is a rule of thumb I have developed based on years of reviewing the performance of Db2 for z/OS systems
  - Objective: configuration with **processing capacity and memory in balance**
  - I have observed in recent years that at many sites, z/OS real storage increases did not keep pace with processing capacity increases
    - If too little memory, performance potential of processors **not fully realized**
- Note that z/OS supports **up to 4 TB** of memory for an LPAR
  - Largest real storage size I’ve seen for an LPAR with my own eyes: **1.1 TB**



# More about RMF's "N" number of engines

- Information below is from same report as information on previous slide

---CPU---		----- TIME % -----			
NUM	TYPE	ONLINE	LPAR BUSY	MVS BUSY	PARKED
0	CP	100.00	87.03	86.85	0.00
1	CP	100.00	77.76	77.68	0.00
2	CP	100.00	83.88	83.78	0.00
3	CP	100.00	87.07	86.91	0.00
4	CP	100.00	76.23	76.14	0.00
5	CP	100.00	76.79	76.71	0.00
6	CP	100.00	80.45	80.35	0.00
7	CP	100.00	73.29	73.24	0.00
8	CP	100.00	63.83	69.22	0.00
9	CP	100.00	57.78	62.95	0.00
A	CP	100.00	35.28	48.33	17.01
TOTAL/AVERAGE			72.67	75.16	
12	IIP	100.00	66.63	58.68	0.00
				46.30	0.00
13	IIP	100.00	26.70	23.42	0.00
				18.24	0.00
14	IIP	100.00	9.21	8.07	0.00
				6.42	0.00
3E	IIP	100.00	0.00	-----	100.00
				-----	100.00
TOTAL/AVERAGE			25.64	26.86	

10.8 general-purpose engines – 0.8 part reflects fact that CPU A is 17% parked from perspective of this LPAR, meaning that LPAR has access to 80% of the engine's processing capacity (round 17% up to 20%)

- 3 zIIP engines (engine 3E is 100% parked for LPAR)
  - 3 zIIP engines are running in SMT2 mode (indicated by fact that each engine has 2 values in MVS BUSY column), so they count as 6 "places where work can be dispatched"
  - That's why N = 16.8 on previous slide: 10.8 general-purpose engines (they always run in "uni-thread" mode), and 6 zIIP "places to dispatch work"
  - "At least 20 GB of memory per engine" refers to engine "cores," so I use 3 as number of zIIPs, not 6

Recommended minimum memory amount is (10.8 + 3) = 13.8, round that to 14, times 20 GB, equals 280 GB

# Whatever amount of memory an LPAR has...

- ...how much can you use?
- Don't want too much pressure on LPAR's real storage resource
- Here is my rule of thumb regarding real storage usage: go ahead and use more to boost Db2 for z/OS performance, as long as LPAR's **demand paging rate** doesn't get out of hand
  - That's the rate, per second, at which pages that z/OS moved from real to auxiliary storage are brought back into real storage on demand
  - If demand paging rate is **less than 1 per second, it's not out of hand**
    - Very small but non-zero paging rate is NOT something that would concern me
  - I would not want demand paging rate to go beyond 2-3 per second
    - If it's at that level, I'd add more memory before using more memory



# Where can I find LPAR's demand paging rate?

- Check RMF Summary report for the LPAR

"No pressure on storage here!"



NUMBER OF INTERVALS 4			TOTAL LENGTH OF INTERVALS 00.59.58															
-DATE	TIME	INT	CPU	DASD	DASD	TAPE	JOB	JOB	TSO	TSO	STC	STC	ASCH	ASCH	OMVS	OMVS	SWAP	DEMAND
MM/DD	HH.MM.SS	MM.SS	BUSY	RESP	RATE	RATE	MAX	AVE	MAX	AVE	MAX	AVE	MAX	AVE	MAX	AVE	RATE	PAGING
11/03	09.15.00	15.00	51.5	0.4	35515	36.5	83	72	96	92	371	365	0	0	11	6	0.00	0.00
11/03	09.30.00	14.59	51.8	0.4	34656	81.8	85	68	98	95	369	362	1	0	11	5	0.00	0.00
11/03	09.45.00	15.00	49.4	0.4	34461	70.7	75	68	95	92	363	359	0	0	12	4	0.00	0.00
11/03	10.00.00	14.59	52.1	0.4	39537	288.0	82	70	94	91	365	358	0	0	15	4	0.00	0.00
-TOTAL/AVERAGE			51.2	0.4	36042	119.3	85	69	98	93	371	361	1	0	15	5	0.00	0.00

**Buffer pools – your single biggest  
memory-related leverage point**

# The most important buffer pool metric

- Total read I/O rate

- All synchronous + all prefetch reads for pool, per second
  - 3 categories of prefetch read: sequential, list and dynamic
- Once source of numbers: Db2 monitor statistics long report
  - Depending on monitor, may be called statistics detail report

BP3	READ OPERATIONS	QUANTITY	/SECOND
-----	-----	-----	-----
	SYNCHRONOUS READS	1676.1K	<b>465.56</b>
	SYNCHRON. READS-SEQUENTIAL	375.00	0.10
	SYNCHRON. READS-RANDOM	1675.7K	465.45
	GETPAGE PER SYN.READ-RANDOM	18.36	
	SEQUENTIAL PREFETCH REQUEST	56125.00	15.59
	SEQUENTIAL PREFETCH READS	9912.00	<b>2.75</b>
	PAGES READ VIA SEQ.PREFETCH	9912.00	2.75
	S.PRF.PAGES READ/S.PRF.READ	1.00	
	LIST PREFETCH REQUESTS	4309.00	1.20
	LIST PREFETCH READS	191.00	<b>0.05</b>
	PAGES READ VIA LIST PREFETCH	2370.00	0.66
	L.PRF.PAGES READ/L.PRF.READ	12.41	
	DYNAMIC PREFETCH REQUESTED	179.9K	49.96
	DYNAMIC PREFETCH READS	26440.00	<b>7.34</b>

For this buffer pool, total read I/O rate is  
 $465.56 + 2.75 + 0.05 + 7.34 = 475.7/\text{second}$

# Can also get numbers via Db2 command

- -DISPLAY BUFFERPOOL(ACTIVE) DETAIL

- Issue command, wait one hour, then issue command again
- In output of second issuance of command, divide counters by 3600 to get per-second figures
  - Note: in command output, synchronous reads are in two categories: random (R) and sequential (S) – add these two numbers to get total synchronous reads

```
DSNB401I  -DBP1 BUFFERPOOL NAME BP2

          SYNC READ I/O (R) =31107574
          SEQ.   GETPAGE    =133932496
          SYNC READ I/O (S) =664918
          SYNC READ I/O (ZHL) =0
          DMTH HIT          =0
          PAGE-INS REQUIRED  =0
          SEQUENTIAL        =1110191
          VPSEQT HIT        =822222
          RECLASSIFY        =14131538
DSNB412I  -PROD SEQUENTIAL PREFETCH -
          REQUESTS          =650334
          PREFETCH I/O     =385115
          PAGES READ        =20531190
DSNB413I  -PROD LIST PREFETCH -
          REQUESTS          =1641349
          PREFETCH I/O     =160339
          PAGES READ        =1527361
DSNB414I  -PROD DYNAMIC PREFETCH -
          REQUESTS          =10229380
          PREFETCH I/O     =2917561
```

For this buffer pool, total read I/O rate is (31,107,574 + 664,918 + 385,115 + 160,339 + 2,917,561) divided by 3600, which is 9787/second

# Your primary aim in buffer pool tuning

- Drive total read I/O rates as low as you can by enlarging pools with higher read I/O rates – but don't over-burden system memory
  - Remember: ideal is demand paging rate that is less than 1 per second
- Reducing read I/Os will improve response time (less I/O wait time) and save CPU (every I/O consumes some CPU time)



# Measuring results of buffer pool tuning

- Use Db2 monitor accounting long reports (may be called accounting detail reports) generated **before and after** buffer pool change
  - Reports should cover same time period of same day of week (i.e., 9-10 AM, Monday)
  - I like report data to be **aggregated at connection type** level
    - Depending on monitor, may involve telling monitor to “order” or “group” data by connection type
    - One report section shows CICS-Db2 activity, another shows DDF activity, etc.
  - Look for reductions in these “average” values: synchronous database read wait time, “other” read wait time (prefetch read) and class 2 (i.e., in-Db2) CPU time (“average” is generally per transaction or per batch job, depending on workload)

# Driving down buffer pool read I/O rates

- Generally want to focus on pools with highest read I/O rates
  - If you have any buffer pools with read I/O rates > 1000/second, maybe try to get those below 1000/second
  - Some organizations that super-size buffer pools are aiming for read I/O rates < 100/second, or even < 10/second for each pool
- Some data points:
  - Highest total read I/O rate I have seen for a buffer pool: 12,144 per second (average over a 1-hour period)
  - **Largest buffer pool configuration** I have seen for a Db2 subsystem (aggregate size of all pools): **879 GB**
    - This on an LPAR with 1104 GB of real storage – demand paging rate was zero
  - **Largest individual buffer pool** I have seen: **297 GB**

# If Db2 subsystem part of a data sharing group...

- ...don't forget about **group buffer pools** (GBPs)
- Enlarging buffer pool BP2 (for example) for members of data sharing group **may necessitate enlarging GBP2** in coupling facility
- If local buffer pools too large relative to group buffer pool, could see **directory entry reclaims** for GBP – a drag on performance
  - If GBP is way undersized relative to associated local buffer pools, could even see **GBP write failures due to lack of storage** – that could lead to pages going on logical page list (LPL), perhaps causing program failures
  - Can check on directory reclaims and write failures due to lack of storage using output of this command:

```
-DISPLAY GROUPBUFFERPOOL(*) TYPE(GCONN) GDETAIL(INTERVAL)
```



# Buffer pool tuning not just about pool size

- Recommendation: use **large real storage page frames** to back any pool that has a **GETPAGE rate > 1000 per second**
  - A measure of buffer pool activity – a GETPAGE is a Db2 request to access a page
    - Get GETPAGE rate from Db2 monitor statistics long report, or from output of -DISPLAY BUFFERPOOL(ACTIVE) DETAIL (add figures for random and sequential GETPAGEs)
  - Large page frames save CPU by making **translation** of virtual storage addresses to real storage addresses **more CPU-efficient**
  - To be backed by large page frames, buffer pool must be **page-fixed in memory** – done via -ALTER BUFFERPOOL with PGFIX(YES)
  - Also need to **request** large frames for pool via -ALTER BUFFERPOOL with **FRAMESIZE(1M)** or **FRAMESIZE(2G)**, for 1 MB or 2 GB frames, respectively

# More on large page frames

- For existing pool, change from PGFIX(NO) to PGFIX(YES) happens when pool is subsequently deallocated and reallocated (usually as consequence of stopping and restarting Db2 subsystem)
  - Same is true for change in pool's frame size
- To verify that pool is backed by large page frames, look at output of -  
DISPLAY BUFFERPOOL(ACTIVE) DETAIL

```
DSNB401I  -DBP1 BUFFERPOOL NAME BP2
DSNB402I  -DBP1 BUFFER POOL  SIZE = 3700000 BUFFERS
DSNB546I  -DBP1 PREFERRED FRAME SIZE 2G ←
          3670016 BUFFERS USING 2G FRAME SIZE ALLOCATED
DSNB546I  -DBP1 PREFERRED FRAME SIZE 2G
          29984 BUFFERS USING 1M FRAME SIZE ALLOCATED ←
```

Pool is defined with FRAMESIZE(2G)

524,288 buffers of 4 KB each will fill one 2 GB frame  
– “left over” buffers will be backed by 1 MB frames

If pool has preferred frame size of 2G or 1M and command output shows that some buffers not backed by large frames, it means system does not have enough large frames to fully back pool – can be addressed by changing value of LFAREA parameter in IEASYSxx member of system's SYS1.PARMLIB data set

# New with Db2 12: contiguous buffer pools

- A contiguous buffer pool is one defined in a Db2 12 system with a specification of `PGSTEAL(NONE)`
  - `PGSTEAL(NONE)` means buffer stealing not *expected* to occur for pool, because it has enough buffers to hold all pages of all assigned objects
- At first access of object assigned to `PGSTEAL(NONE)` pool, requested page(s) brought into memory immediately, and then **all the rest of object's pages will be asynchronously read into pool**
  - Not only that, but pages will be **arranged in memory as they are on disk** (thus the term “contiguous”) – result is **more CPU-efficient page access**
- Note: **2 GB page frames will not be used** for `PGSTEAL(NONE)` buffer pool – if you want pool to be backed by large frames, **specify `FRAMESIZE(1M)`**

# More on Db2 12 contiguous buffer pools

- Even for PGSTEAL(NONE) pool, buffer stealing will occur if necessary (i.e., if pool full and page needs to be brought into pool)
- To allow for buffer stealing while maintaining “contiguous-ness,” **part of pool designated as steal area** – any required buffer stealing happens there, and pages in contiguous area will not be disturbed
  - Buffers in steal area managed using FIFO (first in, first out) steal algorithm
  - Size of steal area will be **10% of pool**, though **not more than 6400 buffers** and **not less than 50 buffers**
- For max CPU efficiency benefit of contiguous pool, want **pool’s steal area to be empty** (i.e., all pages fit in contiguous part of pool)
  - So, if PGSTEAL(NONE) pool has 50,000 buffers, want total pages of objects assigned pool to be  $\leq 45,000$

# **Other ways to exploit real storage to enhance Db2 performance**

# RELEASE(DEALLOCATE) packages

- For package bound with RELEASE(DEALLOCATE), the package and any *table space-level* (or partition-level) locks acquired in executing package **will remain allocated to thread**, until thread deallocated
  - Versus package, table space locks being released from thread at commit
- When RELEASE(DEALLOCATE) package executed with **persistent thread** (persists through commits), can get **significant CPU savings** from retaining the package and table space locks for life of thread
  - Eliminate cost of releasing, reacquiring package and associated table space locks after each commit
- **Memory angle:** when RELEASE(DEALLOCATE) packages executed via persistent threads, more virtual and real storage used for threads

## More on RELEASE(DEALLOCATE)

- **Examples of persistent threads:** CICS-Db2 protected entry threads, high-performance DBATs, and threads used by batch jobs
- For transactional applications, best use of RELEASE(DEALLOCATE) is for packages **executed frequently and via persistent threads**
  - CPU efficiency benefit tends to be greater for packages that have **very low in-Db2 CPU time** per execution (e.g., less than 10 ms)
- RELEASE(DEALLOCATE) package can also deliver performance benefit for **batch job that issues many commits**
- Note: “regular” DBAT becomes a high-performance DBAT when used to execute RELEASE(DEALLOCATE) package **and** when **PKGREL = BNDOPT** for DDF (versus PKGREL = COMMIT)
  - To check PKGREL setting, issue -DISPLAY DDF DETAIL

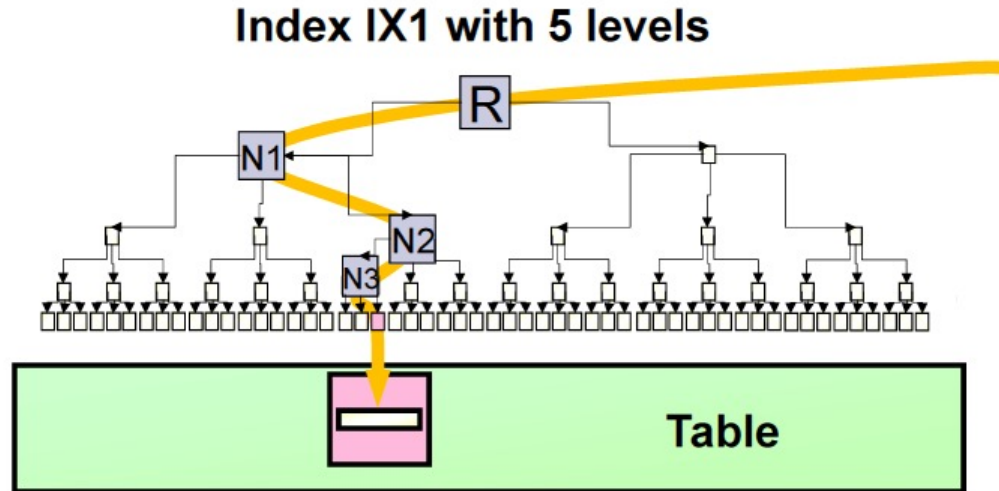
## And a bit more...

- One way to get high-performance DBATs: bind **IBM Data Server Driver / Db2 Connect** packages with `RELEASE(DEALLOCATE)`
  - Do not bind those packages with `RELEASE(DEALLOCATE)` in default NULLID collection – that would make all DBATs high-performance by default
    - Instead, **BIND COPY** NULLID packages with `RELEASE(DEALLOCATE)` into **alternate collection** and point selected DDF-using applications to alternate collection using **Db2 profiles tables**, as described in this blog entry:  
<http://robertsdb2blog.blogspot.com/2018/07/db2-for-zos-using-profile-tables-to.html>
- Historically, combination of `RELEASE(DEALLOCATE)` packages and persistent threads could interfere with package rebind actions
  - `RELEASE(DEALLOCATE)` “**break in**” feature of Db2 11 helped with that
  - The **rebind phase-in** feature of Db2 12 function level 505 helps even more



# Db2 12 index fast traverse blocks (FTBs)

- As table grows, indexes defined on table go to additional levels
- Each level added to index adds a GETPAGE to every index “probe”
- More GETPAGEs lead to increased CPU cost of query execution

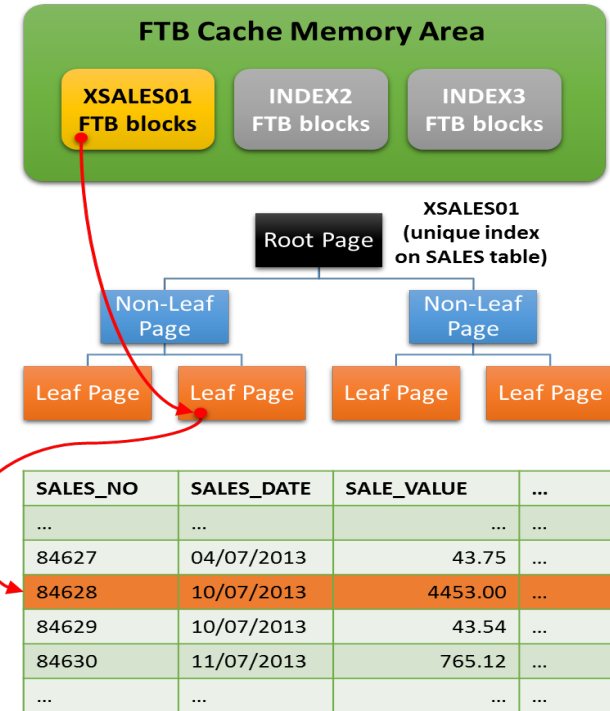


Accessing table row via 5-level index requires 6 GETPAGEs – 5 of which are index-related

# How FTBs improve CPU efficiency

- Db2 12 can build FTB structures for certain indexes (function level 508 + PH30978: non-unique OK)
- How an FTB improves CPU efficiency:
  - Suppose query has “equals” predicate that matches an index for which an FTB structure has been built by Db2
  - Db2 can go to FTB with value specified in predicate, and FTB identifies index leaf page in which value is located
  - So, for 5-level index, instead of needing 6 GETPAGEs to access data row (5 index, 1 table space), now only need 2 GETPAGEs (1 for index leaf page, 1 for table space)
- **Default:** amount of DBM1 space that can be used for FTBs limited to **20% of buffer pool configuration size**
  - If 40 GB buffer pool configuration, max FTB space = 8 GB

```
SELECT SALES_DATE, SALE_VALUE ...  
FROM SALES  
WHERE SALES_NO = 84628
```




# Dynamic statement caching

- Global statement cache has been above 2 GB bar in DBM1 address space since Db2 V8
  - Size determined by ZPARM parameter EDMSTMTC
- Larger statement cache = more cache “hits”
  - CPU savings achieved through resulting **avoidance of full PREPAREs**
- I regularly see dynamic statement cache hit ratios of **95% or higher** – aim for that on your system (can check on dynamic statement cache hit rate with Db2 monitor statistics long report)

# The EDM pool: package and DBD caches

From Db2 monitor  
statistics detail report  
(or online display)



EDM POOL	QUANTITY	/SECOND
-----	-----	-----
PAGES IN DBD POOL (ABOVE)	150.0K	N/A
FREE PAGES	49076.25	N/A
FAILS DUE TO DBD POOL FULL	0.00	0.00
PAGES IN SKEL POOL (ABOVE)	200.0K	N/A
FREE PAGES	75860.15	N/A
FAILS DUE TO SKEL POOL FULL	0.00	0.00
DBD REQUESTS	11264.9K	3182.17
DBD NOT FOUND	12.00	0.00
DBD HIT RATIO (%)	100.00	N/A
PT REQUESTS	39556.5K	11.2K
PT NOT FOUND	107.00	0.03
PT HIT RATIO (%)	100.00	N/A

Definitely  
want zeroes  
here

- For both DBD cache and package cache, you want REALLY high ratio of “requests” to “not found” – like thousands to one (“not found” requires read from directory)
- If that ratio is not really high for one of these caches, make the cache larger (in ZPARM, it's EDMDBDC for DBD cache, and EDM\_SKELETON\_POOL)

# Data sets: are you hitting the DSMAX limit?

- A DSMAX value set years ago **may be too small today**
  - Result can be a high rate of physical closure of Db2 data sets
    - Not good – data set physical open/close operations are **relatively expensive**
  - If you see (in a Db2 monitor statistics report) a lot of data set close activity due to the DSMAX threshold being reached, **increase DSMAX**
    - 200,000 is max value, but practical limit probably less due to below-the-bar DBM1 virtual storage consumption – up to 70,000 probably OK in most cases
    - Don't go overboard – good value is just large enough to make rate of data set close actions due to threshold reached **either zero or very small**

OPEN/CLOSE ACTIVITY	QUANTITY	/SECOND
OPEN DATASETS - HWM	13698.00	N/A
OPEN DATASETS	13313.75	N/A
DSETS CLOSED-THRESH.REACHED	0.00	0.00

- Zero is good
- A few per hour is OK
- Several per minute not so good

# MXDTCACH

- In executing query, Db2 can build [sparse index](#) on data materialized in work file – improves efficiency for repeated access to that data
- MXDTCACH parameter in ZPARM specifies amount of memory that can be used for sparse index for a given process (default is 20 MB)
- In Db2 monitor statistics long report:

MISCELLANEOUS	AVERAGE	TOTAL
-----	-----	-----
SPARSE IX BUILT WF	0.36	8

Number of times sparse index was built in work file, because not enough space to build it in memory

- If non-zero and you have memory to spare, consider making MXDTCACH larger (e.g., 30 MB or 40 MB) to allow more sparse indexes to be built entirely in memory (boosts CPU efficiency of query execution)

# Sort pool

- Space in DBM1 (above the 2 GB bar since Db2 V8) that is used for **SQL-related sorts** (as opposed to utility sorts)
- The larger the sort pool, the more **CPU-efficient** SQL sorting can be
  - Size determined by ZPARM parameter SRTPOOL
  - Note that this is the maximum size of the sort work area that Db2 will allocate *for each concurrent sort user*
    - Default size of sort pool is 10 MB
    - Max SRTPOOL value is 128 MB – largest I've seen is 48 MB
    - Db2 12: **sort tree has MANY more nodes** than before, so larger SRTPOOL more likely to increase level of in-memory sorting

# Robert Catterall

[rfcatter@us.ibm.com](mailto:rfcatter@us.ibm.com)