(and 8 and 9)

Exploiting DB2 10 for z/OS CPU-saving technology

Robert Catterall

DB2 for z/OS Specialist

IBM

This presentation, in a nutshell

- Every release of DB2 for z/OS provides multiple new capabilities that you can leverage to improve system and application performance
- At lots of sites, many of the performance-enhancing features delivered in recent DB2 releases are not being fully exploited
- Presented here is information aimed at helping people to take advantage of performance-boosting technology introduced with DB2 for z/OS Versions 8, 9 and 10

Agenda

- Buffer pool configuration optimization techniques
- Leveraging Big Memory for improved DB2 CPU efficiency
- Making your index configuration leaner and meaner
- Improving application execution efficiency at the thread level
- Other CPU-efficiency boosters

Buffer pool configuration optimization techniques

Page-fixing buffers (DB2 V8)

- To implement: -ALTER BUFFERPOOL (*bpname*) PGFIX(YES)
  - Change doesn't take effect immediately – requires deallocation and reallocation of the pool
    - Usually associated with STOP/START of the DB2 subsystem, but could be done by altering VPSIZE of pool down to 0 and then back to former value
  - Default is PGFIX(NO)
- With PGFIX(YES) in effect, pool's buffers are fixed in memory (i.e., they can't be stolen and paged out to auxiliary storage)
  - Benefit: DB2 won't have to ask z/OS to fix a buffer in memory (and then release it) every time a page is read into or written from the buffer
    - 1 prefetch read: maybe 32 (or more) instances of page-fix/page-release
    - Applies as well to writes to/reads from group buffer pools, if data sharing
  - Reduction in page-fix/page-release activity could reduce CPU consumption of DB2-accessing programs by several percentage points

Don’t overdo page-fixing of DB2 buffers

- You might think, “Hey, why not page-fix all of my buffer pools?”
  - That may or may not be a good idea
    - If running DB2 V8 or V9, benefit of PGFIX(YES) for low I/O pools (those with total read I/O rate* < 100 per second) would be marginal (DB2 10 adds a new benefit – see next slide)
    - Don’t want to make too much of an LPAR’s real storage resource non-pageable (at least one organization uses PGFIX(YES) for all pools, but real storage on associated LPAR is 4X larger than aggregate buffer pool size)

* Total read I/O rate for pool = total synchronous reads (random + sequential) plus total prefetch reads (list + dynamic + sequential), per second
  - Get numbers from DB2 monitor statistics report or online display
  - Another source: issue command -DISPLAY BUFFERPOOL(ACTIVE) DETAIL, then issue it again one hour later, and divide numbers in output of second issuance of the command by 3600 to get per-second figures

DB2 10: a new PGFIX(YES) benefit

- Starting with z10 servers and z/OS 1.10, a portion of mainframe memory can be managed in 1 MB page frames
- DB2 10 will use 1 MB page frames to back a PGFIX(YES) buffer pool (if not enough 1 MB page frames to fully back pool, DB2 10 will use 4 KB frames in addition to 1 MB frames)
  - Bigger page frames means more CPU-efficient translation of virtual-to-real storage addresses
  - This CPU benefit is on top of the previously mentioned savings resulting from less-costly I/Os into and out of page-fixed buffer pools
- DB2 V8 and V9 can't use 1 MB page frames for buffer pools
  - To be on safe side, when going to DB2 10 don't configure an overly large part of server memory to be managed with 1 MB page frames until you've completed migration and are likely past need for fallback

"Pinning" objects in memory (pre-DB2 10)

- By default, DB2 utilizes a least-recently-used (LRU) algorithm for buffer stealing (i.e., to select occupied buffers that will be overwritten to accommodate new pages brought in from disk)
  - And, this is the right algorithm to use in most cases
- Pre-DB2 10 system: if you want to use a buffer pool to "pin" database objects in memory (i.e., to have those objects cached in memory in their entirety), use the FIFO buffer steal algorithm
  - Why? Because tracking buffer usage on a FIFO basis (first in, first out) is simpler than LRU and so uses less CPU
    - If objects are cached in their entirety (i.e., if total number of pages for objects assigned to pool is less than number of buffers in pool), there won't be any buffer stealing, so why spend CPU tracking buffer usage?
  - Implement with -ALTER BUFFERPOOL (*bpname*) PGSTEAL(FIFO)

DB2 10: a better option for object-pinning

- New buffer steal algorithm for pools: PGSTEAL(NONE)
- When PGSTEAL(NONE) is specified for a pool:
  - When an object (table space or index) assigned to the pool is first accessed, requesting application process gets what it needs and DB2 asynchronously reads all the rest of the object's pages into the pool
  - In optimizing SQL statements that target the object that is now pinned in the buffer pool, DB2 will assume that I/Os will not be required
  - If objects assigned to a PGSTEAL(NONE) pool have more pages than the pool has buffers, DB2 automatically switches to FIFO buffer steal algorithm to accommodate new page reads from disk when pool is full

Leveraging Big Memory for improved DB2 CPU efficiency

What is “Big Memory?”

- Refers to the real and virtual storage resource available with 64-bit addressing (first exploited by DB2 for z/OS V8)
  - 16 million terabytes of addressability
- Real storage sizes are getting bigger and bigger
  - z196, zEC12 servers can be configured with up to 3 TB of memory
  - Production z/OS LPARs with real storage resources of 40 GB or more are increasingly common (seeing some with 100+ GB of real storage)
- Many organizations are not effectively leveraging Big Memory
  - Put your system’s idle gigabytes to work (DB2 can help)
  - In doing that, keep an eye on your system’s demand paging rate (available via a z/OS monitor) – don’t let that get out of hand*

* If the z/OS LPAR’s demand paging rate is in the low single digits per second or less, it’s not out of hand

Bigger buffer pools

- Buffer pools went above the 2 GB “bar” in DBM1 with DB2 V8
  - So, you’re not going to run out of virtual storage – key to accommodating larger buffer pools is having enough real storage to back them
- Buffer pool enlargement: key criterion is the total read I/O rate per second for a pool (see slide 6)
  - If this rate is over 1000 per second for a pool, definitely consider enlarging the pool if the LPAR’s real storage resource is adequate
    - Larger pool = fewer I/Os = reduced CPU consumption
  - Do you have enough real storage to support a buffer pool size increase?
    - If LPAR’s demand paging rate is in the low single digits per second (or less), memory should be sufficient for a buffer pool size increase

More on big buffer pools

- If no pools have total read I/O rate > 1000/second (or if adding buffers to high-I/O pools is showing diminishing returns), may want to enlarge some pools that have I/O rates > 100/second
  - Pools with total read I/O rate of < 100/second generally not a concern
- Even if LPAR's demand paging rate is very low, be careful about using > 50% of an LPAR's memory for DB2 buffer pools
  - An organization utilizing WLM-managed DB2 buffer pool sizing reported seeing their buffer pool configuration settle in at 30-40% of LPAR memory (ALTER BUFFERPOOL(*bpname*) AUTOSIZE(YES))
- A data point: a site had a pool with a read I/O rate of 9000/second
- Another data point: an organization has a DB2 for z/OS subsystem with a 46 GB buffer pool configuration (aggregate size of all pools allocated for that subsystem)

DB2 data sharing? Don't forget GBPs

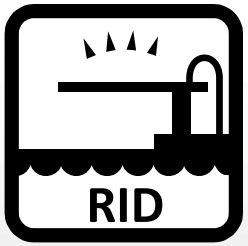
- Referring here to group buffer pools in the coupling facilities
- If you make BPn larger across members of a data sharing group, you may need to enlarge GBPn, as well
  - If aggregate size of local BPs too large relative to size of corresponding GBP, might see many directory entry reclaims – drag on performance
  - Can check on directory entry reclaim activity using output of DB2 command -DISPLAY GROUPBUFFERPOOL(*gbpname*) GDETAIL
  - A sizing guideline for 4K buffer pools: when ratio of directory entries to data entries in GBP is default 5 to 1, directory entry reclaims likely to be zero if size of GBP is 40% of combined size of associated local BPs
    - Example: 3-way group, BP1 at 40K buffers (160 MB) on each member
    - Good GBP1 size is 40% X (3 X 160 MB) = 192 MB
    - Blog entry with more info: http://robertsdb2blog.blogspot.com/2013/07/db2-for-zos-data-sharing-evolution-of.html

Leveraging Big Memory: statement caching

- Global dynamic statement cache “above the bar” since DB2 V8, so (as with buffer pools) concern is real storage utilization
- Larger statement cache = more cache “hits” = CPU savings
  - In many cases, you should be able to achieve a hit ratio > 90%
- DB2 10 can boost dynamic statement cache hit ratio by, in effect, parameterizing dynamic statements containing literals
  - Use CONCENTRATE STATEMENTS WITH LITERALS with PREPARE (or enable via keyword in client data source or connection property)
  - If match for dynamic statement with literals not found in cache, literals replaced with & and cache is searched to find match for new statement
    - If not found, new statement is prepared and placed in the cache
  - Note: for some predicates, you may WANT DB2 to optimize using literal values (range predicates can be in this category)

DB2 10: more space for processing RIDs

- DB2 processes lists of RIDs (i.e., row IDs – from index entries) as a consequence of several access paths, including:
  - List prefetch
  - Multi-index access (aka index ANDing and index ORing)
- CPU savings can be achieved if RID pool is large enough to enable RID processing for a query to complete in memory
  - RID pool size determined by value of MAXRBLK parameter in ZPARM
  - With DB2 10, default size of RID pool went from 8 MB to **400 MB** (RID pool has been above the 2 GB bar in DBM1 since DB2 V8)
    - Note: you're likely to see more list prefetch activity in a DB2 10 environment versus prior releases of DB2 (DB2 10 can use list prefetch to read index leaf pages, as well as data pages, into memory)

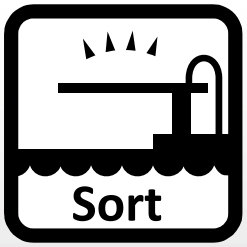


What if your DB2 10 RID pool isn't big enough?

- Suppose there is not enough space in the RID pool to allow a RID list processing operation to complete in memory?
  - Before DB2 10: DB2 will abandon RID list processing and go with a table space scan for data access
  - DB2 10: DB2 continues working on the RID list processing operation, using space in the work file database (except in the case of hybrid join)
  - Though that's not quite as CPU-efficient as getting RID list processing done in memory, it's better than abandoning RID list processing and falling back to a table space scan
  - If RID list processing overflows to a work file table space, you can get CPU savings if the buffer pool dedicated to 32KB-page work file table spaces is large enough to keep read I/O rate down
  - New ZPARM parameter MAXTEMPS_RID can be used to limit amount of work file space that one RID list can use

DB2 10: more space for sorting rows, too

- 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)
  - A larger sort pool means CPU savings for SQL-related sorts
  - Sort pool 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*
    - So, don't go overboard here if you have a lot of concurrent SQL sort activity on your system – you don't want to over-commit real storage
  - With DB2 10, default size of sort pool went to 2 MB to **10 MB**
  - Maximum SRTPOOL value is 128 MB
    - Data point: one site has SRTPOOL set to 48 MB (I/O activity for the work file-dedicated buffer pools is very low at this site – perhaps related to the large sort pool value)



Leveraging Big Memory: that's not all

- Preceding slides described some global (i.e., subsystem-wide) uses of system memory that can deliver CPU savings
- In a subsequent section of this presentation we'll look at some thread-level actions that increase memory utilization and decrease CPU utilization

Making your index configuration leaner and meaner

The benefits of index configuration pruning

- "Pruning:" elimination of indexes that aren't doing you any good
  - Getting rid of such indexes means CPU savings for insert and delete operations, and for updates of indexed columns, and for several utilities (e.g., RUNSTATS, REORG, LOAD)
  - You also get some disk capacity back
- So, which of your existing indexes are just taking up space?
  - That used to be a hard question to answer (can't determine via SYSPACKDEP catalog whether or not an index is used by *dynamic* SQL statements)
  - LASTUSED column of catalog table SYSINDEXSPACESTATS (introduced with DB2 9) makes it easier to identify useless indexes
    - Default value for this column is NULL

Opportunities to make existing indexes useless…

- …and therefore DROP-able

1. Switch to table-controlled from index-controlled partitioning for range-partitioned tables
   - An easy way to do that: for a table's partitioning index, execute an ALTER INDEX statement with the NOT CLUSTER clause
     - CLUSTER/NOT CLUSTER options of ALTER INDEX were introduced with DB2 V8
   - Once you've converted to table-controlled partitioning, you might find that the former partitioning index is now useless (in other words, it was good only for partitioning – not for improving query performance)
     - In that case, DROP IT
     - Make one of the other indexes on the table the clustering index (so that rows will be clustered by that key within the table's partitions)

Making existing indexes useless…

2. Leverage INDEX INCLUDE functionality introduced with DB2 10
   - Allows you to define a UNIQUE index on a key, and then have one or more other columns included in the index's entries
   - Benefit: you can do with one index in a DB2 10 environment what it took two indexes to do in a DB2 9 or DB2 V8 system
   - Example: for table T1, you want UNIQUE constraint on columns C1 and C2, and index-only access for queries referencing C1, C2, C3, and C4
     - Pre-DB2 10:
       - CREATE UNIQUE INDEX IX1 ON T1 (C1, C2)
       - CREATE INDEX IX2 ON T1 (C1, C2, C3, C4)
     - DB2 10 NFM:
       - ALTER INDEX IX1 ADD INCLUDE COLUMN(C3)
       - ALTER INDEX IX1 ADD INCLUDE COLUMN(C4)
       - **DROP INDEX IX2**

Action will place index in REBUILD-pending status

Making existing indexes useless…

3. DB2 10: consider hash-organizing data in some of your tables
   - Idea: DB2 will place rows in the table based on a hash of a unique key
     - *For a query that contains an "equals" predicate that references the table's hash key*, DB2 can retrieve the row with **1 GETPAGE**, without using an index (unless row is in the table space's overflow area)
     - Example (column ACCOUNT_NUM is unique in table ACCOUNTS):

```
ALTER TABLE ACCOUNTS ADD ORGANIZE BY HASH UNIQUE(ACCOUNT_NUM)
```

Subsequent online REORG of table space puts hash organization into effect

     - In addition to providing super-efficient retrieval of individual[1] rows by ACCOUNT_NUM value, hash organization of data can make unique index formerly required on ACCOUNT_NUM column unnecessary – **DROP IT** [2] *(DB2 doesn't need an index to guarantee uniqueness of a table's hash key)*

1 – Hash-organization of data can boost performance for a single-row SELECT with an "equals" predicate that references the hash key, but it is generally NOT good for sequential processing (including sequential processing that can be triggered by singleton SELECT in a DO loop)
2 – Might want to keep that index if you have queries that reference only the column(s) of the index key and involve a scan (versus a probe) of index values

Consider index upgrade vs. index trim

- Instead of reducing the number of indexes on a table, replace a useless index with one that will improve query performance
- DB2 has recently provided new index upgrade possibilities:
  - Index-on-expression (introduced with DB2 9):
    - Create an index on an expression, like this one:

      ```
      CREATE INDEX IX2 ON TABLE_ABC (SUBSTR(COL1,8,3))…
      ```

    - To make this predicate stage 1 and indexable (vs. stage 2, non-indexable):

      ```
      SELECT… WHERE SUBSTR(COL1,8,3) = 'ST1'
      ```

    - Index-on-expression can be very useful for vendor-supplied applications
    - DB2 10: can create an index-on-expression on in-lined part of a LOB column
  - Index on XML column (new with DB2 9):
    - XML data type by itself can greatly improve XML data access performance
    - Indexing an XML column can REALLY speed things up

Consider larger page sizes for some indexes

- DB2 9 introduced the ability to assign indexes to 8K, 16K, and 32K buffer pools (they formerly could only have 4K-sized pages)
- NOT just about index compression
- For indexes with keys that are not continuously-ascending, and which are defined on tables that get a lot of INSERT activity, larger page sizes could deliver a performance benefit
  - Fewer index page splits (particularly advantageous in a data sharing system, as index page splits are more costly in that environment)
- Other potential benefits associated with larger index page sizes:
  - Might reduce number of index levels (fewer GETPAGEs)
  - Faster, more efficient index scans
- 4K page size probably still best for random index read pattern

Improving application execution efficiency at the thread level

RELEASE(DEALLOCATE) + persistent threads

- "Persistent threads" are threads that persist across commits
  - A thread used by a batch job will persist across commits – it is deallocated when the job completes
  - Threads used by online transactions are persistent when reused
    - Reuse of CICS-DB2 threads often boosted via protected entry threads (PROTECTNUM > 0 in a DB2ENTRY CICS resource definition)
  - Thread reuse, by itself, reduces CPU cost for a transactional workload
  - Thread reuse + RELEASE(DEALLOCATE) bind option for packages executed via persistent threads delivers additional CPU savings
    - Table space-level locks and package sections required for program execution are retained until thread deallocation, instead of being released and reacquired at each commit
    - Batch job bonus: greater benefit from dynamic prefetch and index lookaside, as these performance-boosters are "reset" at each commit

RELEASE(DEALLOCATE) and DB2 10

- Prior to DB2 10, packages allocated to threads were copied to the package table in the EDM pool
  - A limited resource, and RELEASE(DEALLOCATE) + persistent threads made it more so (more pages in that part of the pool are non-stealable)
    - If you run out of space for package table, programs start abending
  - Part of the package table space was below the 2 GB bar in DBM1, and that was a constrained area of virtual storage at many sites
- With DB2 10, packages allocated to threads are copied to an area of DBM1 virtual storage that is above the 2 GB bar, *if the packages were bound or rebound in the DB2 10 environment*
  - And, this virtual storage comes from an agent local pool vs. EDM pool
    - So, in addition to having a lot more “headroom” with respect to using RELEASE(DEALLOCATE), a latch formerly needed to manage utilization of EDM pool space for packages is eliminated – good for throughput

Also new with DB2 10: high-performance DBATs

- In a DB2 10 system, when a DBAT is used to execute a package bound with RELEASE(DEALLOCATE), it becomes a high-performance DBAT
  - Before DB2 10: package bound with RELEASE(DEALLOCATE) treated as though bound with RELEASE(COMMIT) if executed via a DBAT
  - High-performance DBAT will stay dedicated to the connection through which it was instantiated (versus going back into the DBAT pool following transaction completion), and can be reused by 200 transactions
    - So, DDF transactions can now get the CPU benefits of RELEASE(DEALLOCATE) + persistent threads
    - After being used for 200 units of work, high-performance DBAT will be terminated to free up resources
    - Because high-performance DBATs deplete the DBAT pool, you will likely want to increase the value of MAXDBAT in ZPARM to compensate

RELEASE(DEALLOCATE) considerations (1)

- DB2 10 gives you more room to use RELEASE(DEALLOCATE), but you don't want to bind ALL of your packages with this option
  - You won't run out of virtual storage, but RELEASE(DEALLOCATE) + persistent threads will increase real storage usage
    - Keep an eye on your system's demand paging rate (see slide 11)
  - Also: RELEASE(DEALLOCATE) + persistent threads can interfere with some DDL and package bind operations
    - DDF: issue -MODIFY DDF PKGREL(COMMIT), and packages executed via DBATs will be treated as though bound with RELEASE(COMMIT)

      -MODIFY DDF PKGREL(BNDOPT) re-enables high-performance DBATs
    - CICS-DB2: consider changing PROTECTNUM to zero during periods when package rebinds and DDL statements need to be executed
  - Don't use RELEASE(DEALLOCATE) for programs that issue LOCK TABLE – exclusive parent lock would be held until thread deallocation
    - And, keep an eye on lock escalation

RELEASE(DEALLOCATE) considerations (2)

- Best uses:
  - Higher-volume transactions – especially those with lower SQL statement execution cost (for these transactions, CPU cost of release and reacquisition of resources at COMMIT is proportionately higher)
  - For batch programs that issue a lot of commits
- What if you want to use high-performance DBATs for your client-server programs that issue dynamic SQL statements, but not for all such programs?
  - Consider binding packages of IBM Data Server Driver (or DB2 Connect) into default NULLID collection with RELEASE(COMMIT), and into another collection with RELEASE(DEALLOCATE)
    - Have higher-volume client-server transactions use that second collection to gain high-performance DBAT performance benefits (collection name can be specified as a data source property on the client side)

Other CPU-efficiency boosters

Native SQL procedures and DRDA requesters

- Introduced with DB2 9, native SQL procedures provide a means of reducing general-purpose CPU consumption for client-server applications (versus calling external stored procedures)
  - Little of the work done by external stored procedure is zIIP-eligible, even when called via DDF (send/receive processing is zIIP-eligible)
  - Whereas an external stored procedure runs under a TCB in a WLM-managed address space, a native SQL procedure executes under the task of the process that called it
    - If caller is DRDA requester, task under which the native SQL procedure will execute is a preemptable SRB, and as a result a substantial portion (up to 60%) of the native SQL procedure's processing will be zIIP-eligible
    - So, going from external to native SQL procedures for a DB2 client-server application reduces general-purpose CP utilization by increasing zIIP utilization

Native SQL procedures in a DB2 10 system

- SQL procedure language (SQL PL – used to code native SQL procedures) performs better in DB2 10 versus DB2 9
  - DB2 10 allows user-defined functions to be written in SQL PL, too
- Consider binding frequently executed native SQL procedure packages with RELEASE(DEALLOCATE)
  - Pair that with high-performance DBATs and you get a client-server application performance boost
- New WITH RETURN TO CLIENT cursor specification allows a "top-level" calling program to fetch result set rows from a cursor declared and opened in a "nested" stored procedure
  - That option is available for external stored procedures, too
  - No longer have to use temporary tables to make stored procedure-generated result sets available to programs "more than one level up"

Leverage DB2 10 LOB enhancements (1)

- LOAD/UNLOAD large LOBs*, along with non-LOB data, directly from/to SYSREC data set (* could do that before, but only for smaller LOBs)
  - The breakthrough: DB2 10 support for variable blocked spanned (VBS) record format for SYSREC data set (enabled via SPANNED YES on utility control statement)
  - Allows input/output records to exceed 32,760 bytes in length
  - Previously, larger LOB values had to be referenced in SYSREC via file reference variables, and loaded from or unloaded to members of a PDS/PDSE or UNIX System Services files (HFS or zFS)
  - VBS SYSREC: orders of magnitude faster than PDS/PDSE, much faster than HFS/zFS
    - Another benefit is simplification: LOB and non-LOB data values are together in the one SYSREC data set

Leverage DB2 10 LOB enhancements (2)

- Consider in-lining LOB values (i.e., storing a portion of each LOB value, or each LOB value in its entirety, in the associated base table row versus a LOB table space)
  - If most of a table's LOB values can be completely inlined, SELECT and INSERT of LOB values can be much faster versus non-inlined case
    - Performance benefit for LOAD and UNLOAD, too
    - And, significant performance benefit for spatial queries (referring to Spatial Support, part of no-charge DB2 Accessories Suite for z/OS)
  - Even if LOB values cannot be completely inlined, ability to create an index-on-expression for inlined portion of CLOB values could help performance of some queries
- If most of the values in a column can't be completely inlined, *or if LOB values are rarely retrieved*, LOB inlining may not be a good choice

Thanks for your time!