PDUG

Kraków, 28-29 May 2025

DESIGNING FOR HIGH PERFORMANCE CONCURRENT INSERT PROCESSING

John Campbell

Triton Consulting

POLISH DB2 USERS GROUP

AGENDA

PDUG

- Objectives
- Key physical design questions
- Typical performance bottlenecks
- Insert free space search steps
- Relevant Db2 features which can help
- War story
- Summary – Key points
- Appendix – Insert CPU cost rough rule of thumb

OBJECTIVES

PDUG

- Understand key physical design questions
- Understand typical performance bottlenecks
- How to design and tune for optimum performance
- How best to apply and use relevant Db2 features

KEY PHYSICAL DESIGN QUESTIONS

PDUG

- Design for maximum throughput performance or space reuse?
- Random key insert or sequential key insert?
- Store data rows in clustering sequence or insert at the end?
- Input records sorted into clustering key sequence?
- What are the indexing requirements for query and are they justified?

TYPICAL PERFORMANCE BOTTLENECKS

PDUG

- Read and Write I/O for index and data - random key insert or sequential key insert?
- CPU resource consumption
- Active log write
- Lock/latch and service tasks waits
- Sequential keys (e.g., timestamp, dumb sequential number)
- Network latency

READ AND WRITE I/O FOR INDEX AND DATA

PDUG

- Random key insert to index
  - N sync read I/Os for each index
    - N depends on # index levels, # leaf pages and buffer pool hit ratio
    - Index read I/O time = N * # indexes * ~ 1-2 ms
  - Sync data read I/O time = ~ 1-2 ms per page (0 if insert at end)
  - Deferred async write I/O for each page
    - ~ 1-2 ms for each row inserted
    - Depends on channel type, device type, I/O path utilization, distance between pages
  - Recommend keeping the number of indexes to a minimum
    - Challenge the need for low value indexes

READ AND WRITE I/O FOR INDEX AND DATA ...

- Sequential insert to end of data set
  - For data row insert and/or ever-ascending or descending index key insert
  - Can eliminate sync read I/O
  - Deferred async write I/O only for contiguous pages
    - ~0.4 ms per page filled with inserted rows
    - Time depends on channel type, device type, I/O path utilization

READ AND WRITE I/O FOR INDEX AND DATA ...

- Recommendations on deferred write thresholds
  - VDWQT = Vertical (dataset level) Deferred Write Threshold
    - Default: when 5% of buffers updated for one dataset, a deferred write will be scheduled
  - DWQT = (buffer pool level) Deferred Write Threshold
    - Default: when 30% of buffers updated, for one dataset, a deferred write will be scheduled
  - Configure and tune for continuous ‘trickle’ write activity in between successive system checkpoints
    - VDWQT and DWQT will typically have to be set lower for intensive insert workloads

READ AND WRITE I/O FOR INDEX AND DATA ...

- Recommendations on deferred write thresholds ...
  - With high deferred write thresholds, write I/Os for data or index entirely resident in the buffer pool can be eliminated except at system checkpoint or at -STOP TABLESPACE time
  - Use VDWQT = 0% for data buffer pool with low hit ratio (1 - 5%) if single thread insert
    - Else set VDWQT = 150 + # concurrent threads (e.g., 100) if sequential insert to the end of the pageset/partition
    - When 250 buffers are updated for this dataset, 128 LRU buffers are scheduled for write
  - Use VDWQT = 0% for sequential index insert
  - Use default if not sure, also for random index insert

DISTRIBUTED FREE SPACE

PDUG

- Use distributed free space – PCTFREE and/or FREEPAGE
  - For efficient sequential read of index
  - For efficient sequential read of data via clustering index
  - To minimize index leaf page split
- Carefully calculate settings
- Default distributed free space
  - 0 FREEPAGE
  - 5% PCTFREE within data page
  - 10% PCTFREE within index page

DISTRIBUTED FREE SPACE ...

PDUG

- For best insert performance
  - Random key insert to index
    - Use non-zero PCTFREE and/or FREEPAGE for index
      - To reduce index leaf page splits
      - For efficient sequential index read
  - Sequential key insert to index
    - Immediately after LOAD, REORG, or CREATE/RECOVER/REBUILD INDEX
      - Use PCTFREE 0% to reduce the number of index pages and possibly index levels by populating each leaf page 100%
  - Use PCTFREE=FREEPAGE=0 for data to reduce both sync read and async write I/Os for each row
    - Possible performance penalty for query in terms of sync single page read I/O when reading multiple data rows via clustering index

INSERT FREE SPACE SEARCH STEPS

PDUG

- Insert performance is a trade off across
  - Minimizing CPU resource consumption
  - Maximising throughput
  - Maintaining data row clustering
  - Reusing space from deleted rows and minimizing space growth
- Warning: insert space search algorithm is subject to change and it does!

INSERT FREE SPACE SEARCH STEPS …

PDUG

- Tradeoff in free space search
  - Insert to the end of pageset/partition
    - To minimize the cost of insert by minimizing
      - Read/write I/Os, getpages, lock requests
  - Search for available space near the optimal page
    - To store data rows in clustering index sequence
    - To store leaf pages in index key sequence
    - To minimize dataset size
  - Search for available space anywhere within allocated area
    - To minimize dataset size
    - Can involve exhaustive space search which is expensive
    - Use large PRIQTY/SECQTY and large SEGSIZE to minimize exhaustive space search

INSERT FREE SPACE SEARCH STEPS ...

PDUG

- For UTS both PBR and PBG, and classic partitioned table space:
  1. Index Manager will identify the candidate page (next lowest key rid) based on the clustering index
     - If page is full or locked, skip to Step 2
  2. Search adjacent pages (+/-) within the segment containing the candidate page
     - For classic partitioned table space it is +/-16 pages
  3. Search the end of pageset/partition without extend
  4. Search the space map page that contains lowest segment that has free space to the last space map page for up to 50 space map pages
     - This is called "smart space exhaustive search"
  5. Search the end of pageset/partition with extend until PRIQTY or SECQTY reached
  6. Perform "exhaustive space search" from front to back of pageset/partition when PRIQTY or SECQTY reached
     - Very expensive i.e., start with space map with lowest segment with free space and then continue all the way through
- For classic segmented table space, steps are very similar except the sequence is different - 1->2->3->5->4->6

INSERT FREE SPACE SEARCH STEPS …

- Each member maintains for each pageset/partition:
  - First/lowest space map page with free space
  - Current space map page with free space
  - Last space map page
- In general, each step has 3 pages for “false lead” threshold and 100 for lock failure threshold
  - In step 3 (search at end of pageset/partition without extend) there is a little more search than just 3 pages “false lead” to tolerate the hot spot case
- For the search step where cannot extend at end of pageset/partition
  - No false lead or lock threshold applied
  - Try every possible available space before failure
  - Search from the first space map page to the end

INSERT FREE SPACE SEARCH STEPS …

- Elsewhere for the “smart exhaustive search” before extend, it will follow the "lowest segment that has free space" going forward 50 space map pages then it will go to extend
  - The 3 pages “false lead threshold” and 100-page lock failures are applied to each space map page
- If there are multiple inserts in the same UR, then 2nd insert will not search the previous repeated unsuccessful space map pages if the previous insert already gone through more than 25 pages

UNIVERSAL TABLESPACE (AND SEGMENTED TABLESPACE)

PDUG

- SEGSIZE
  - General recommendation is to use large SEGSIZE value consistent with size of pageset
    - Typical SEGSIZE value 32 or 64
  - Large SEGSIZE
    - Provides better opportunity to find space in page near by to candidate page and therefore maintain data row clustering
    - Better chance to avoid exhaustive space search
  - Small SEGSIZE
    - Can reduce spacemap page contention
    - But less chance of hitting “false lead threshold” of 3 and looking for space at the end of pageset/partition
      - “False lead” is when spacemap page indicates there is a data page with room for the row, but on visit to the respective data page this is not the case

MAXROWS N

- Optimization to avoid wasteful space search on especially classic partitioned tablespace with fixed length compressed and true variable length row insert
- Must carefully estimate “average” row size and how many “average” size rows will fit comfortably in a single data page
- When MAXROWS n is reached the page is marked full (and taken out of the space search)
- But introduces on going maintenance challenges
  - Could waste space?
  - What happens if data compression is removed?
  - What happens if switch from uncompressed to compressed?
  - What happens when new columns are added?

PARTITIONING

PDUG

- Use page range partitioning by dividing tablespace into partitions by key range
- Depends on somewhat random key to spread insert workload across partitions
- Can reduce logical and physical contention to improve concurrency and reduce performance cost
- Separate index B-tree for each index partition of partitioned index (good for concurrency)
- Only one index B-tree for non-partitioned index (bad for concurrency)
- Over wide partitioning has potential to reduce number of index levels to reduce performance cost

DATA PAGE SIZE

PDUG

- Use large data page size for sequential inserts to
  - Reduce # getpages
  - Reduce # lock requests
  - Reduce # CF requests
  - Get better space use

ACTIVE LOG WRITE

PDUG

- Log data volume
  - From Db2 Statistics Trace, minimum MB/sec of writing to active log can be calculated as
    - # CIs created in active log * 0.004MB / statistics interval in seconds
  - If throughput is a concern
    - Super size OUTBUFF (log output buffer)
    - Consider use of Db2 data compression
    - Use faster DASD device
    - Use zHyperLink protocol

LOCK/LATCH AND SERVICE TASK WAITS

PDUG

- LOCKSIZE
  - Rule of thumb on LOCKSIZE
    - Page lock (LOCKSIZE PAGE|ANY) as design default
    - Especially if sequentially inserting many rows per page
  - Page P-lock contention in active data sharing environment with GBP-dependency
    - Index page update
    - Spacemap page update
    - Data page update when LOCKSIZE ROW

MEMBER CLUSTER

PDUG

- MEMBER-private spacemap and corresponding data pages
- Beneficial in data sharing environment with GBP-dependency to reduce page P-lock and page latch contention especially when data rows are inserted at end of pageset/partition
  - Spacemap page
  - Data page if LOCKSIZE ROW
- Inserted data rows are not clustered by clustering index
  - Instead, data rows stored in available space in member-private area
- May want to consider using LOCKSIZE ROW and larger data page size with MEMBER CLUSTER
  - Better space use
  - Reduce working set of buffer pool pages when page level locking (LOCKSIZE PAGE|ANY) as design default

TRACKMOD NO

PDUG

- Reduces spacemap page contention in active data sharing environment with GBP-dependency
- Db2 will then not track changed pages in the spacemap pages
- Incremental COPY will use LRSN value in each data page to determine whether a page has been changed since the last copy
- Trade-off as potential for degraded performance for incremental COPY because of pageset/partition scan

DB2 LATCH CONTENTION

PDUG

- Latch counters LC01–32 in Db2 OMPE Statistics Report Layout Long
- Rule-of-thumb on Internal Db2 latch contention rate
  - Investigate if > 10K/sec
  - Ignore if < 1K/sec
- Class 6 for latch for index tree P-lock due to index leaf split (data sharing only)
  - Index split is painful in data sharing as results in 2 forced physical log writes
  - Index split time can be significantly reduced by using faster device for Db2 active log
  - Index splits in random insert can be reduced by providing non-zero PCTFREE

DB2 LATCH CONTENTION ...

PDUG

- Class 19 for logical log write latch, both data sharing and non-data
  - Use LOAD LOG NO instead of SQL INSERT
  - Eliminate Unavailable Log Output Buffer condition
- If > 1K-10K contentions/sec, disabling Accounting Class 3 trace helps to significantly reduce CPU time as well as elapsed time

SERVICE TASK WAITS

PDUG

- Service task waits most likely for preformatting
  - Shows up in Dataset Extend Wait in Accounting Class 3 trace
  - Typically up to 1 second each time, but depends on allocation unit/size and device type
  - Anticipatory and asynchronous preformatting performed by Db2
  - Can be eliminated by LOAD/REORG with PREFORMAT option and high PRIQTY value
    - Do not do this on table space defined with MEMBER CLUSTER

IDENTITY COLUMN AND SEQUENCE OBJECT

PDUG

- In both cases, Db2 to automatically generate a guaranteed unique number for sequencing each row inserted into table
- Much better concurrency, throughput and response time possible
  - Compared to application maintaining a sequence number in one row table, which force serialization (one transaction at a time) from update to commit
  - Potential for 5 to 10 times higher insert/commit rate
- Option to cache (default of 20), saving Db2 Catalog update of maximum number for each insert
  - Eliminating GBP write and log write force for each insert in data sharing environment with GBP-dependency
- Recycling or wrapping of identity column and sequence

MULTI ROW INSERT (MRI)

PDUG

- INSERT INTO TABLE for N Rows Values (:hva1,:hva2,...)
- Up to 40% CPU time reduction by avoiding SQL API overhead for each INSERT call
  - % improvement lower if more indexes, more columns, and/or fewer rows inserted per call
- ATOMIC (default) is better from performance viewpoint as create of multiple SAVEPOINT log records can be avoided
- Dramatic reduction in network traffic and response time possible in distributed environment
  - By avoiding message send/receive for each row
  - Up to 8 times faster response time and 4 times CPU time reduction

LARGE INDEX PAGE SIZE

PDUG

- Potential to reduce the number of index leaf page splits, which are painful especially in data sharing environment with GBP-dependent index
  - Reduce index tree lotch contention
  - Reduce index tree P-lock contention
- Potential to reduce the number of index levels
  - Reduce the number of getpages
  - Reduce CPU resource consumption
- Possibility that large index page size may aggravate index buffer pool hit ratio for random access
  - Increase VPSIZE for index buffer pool to compensate

IDENTIFYING UNREFERENCED INDEXES

PDUG

- Additional indexes incur overhead for
  - Data maintenance
    - INSERT, UPDATE, DELETE
  - Utilities
    - REORG, RUNSTATS, LOAD etc.
  - Query optimization
    - Increase number of choices for Db2 Optimizer to consider
- But identifying unused indexes is a difficult task
  - Especially in a dynamic SQL environment

IDENTIFYING UNREFERENCED INDEXES …

- Real Time Statistics (RTS) records the last used date Data maintenance
  - SYSINDEXSPACESTATS.LASTUSED
    - Updated once in a 24-hour period
    - RTS service task updates at first externalization interval (set by STATSINT) after 12pm
    - If index is used by Db2, update occurs
    - If the index was not used, no update
- "Used" as defined by Db2 means
  - As an access path for query or fetch
  - For searched UPDATE / DELETE SQL statement
  - As a primary index for referential integrity
  - To support foreign key access

TABLE APPEND OPTION

PDUG

- CREATE / ALTER TABLE … APPEND YES
- Always use with MEMBER CLUSTER in data sharing environment
- Will reduce search of very long chain of spacemap pages as table space keeps getting bigger
- But will drive need for frequent table space reorganization if data row clustering is required for query performance
- Degraded query performance until the reorganization is performed
- Behavior the same as “pseudo append” with “MC00”
  - MEMBER CLUSTER and PCTFREE=FREEPAGE=0
  - Will switch between append and insert mode
  - Success depends on deletes and inserts being spread across Db2 members of data sharing group

WAR STORY ABOUT INSERT PROCESSING WITH MEMBER CLUSTER

PDUG

- Customer case study scenario
  - PBG UTS, MAXPARTITIONS 2, no MEMBER CLUSTER, random insert, LOCKSIZE ROW, PCTFREE at 80%
  - Many concurrent insert threads
  - Each application commit scope has multiple inserts (10-15) and updates
  - Application groups multiple rows with similar key for related business transaction into single commit scope
  - Data rows frequently archived out of the table
  - Frequent REORG to re-establish PCTFREE to accommodate random insert and to avoid update creating relocated rows
- Everything working well …
  - Until DBA decided to introduce MEMBER CLUSTER to reduce space map page contention whilst still keeping PCTFREE at 80%, LOCKSIZE ROW and frequent REORG
  - Pre-production testing performed to prove the change, but not at production level stress in terms of concurrency

WAR STORY ABOUT INSERT PROCESSING WITH MEMBER CLUSTER

PDUG

- What happens next
  - After REORG to introduce MEMBER CLUSTER, the table space immediately grows into the 2nd partition
    - During peak period, all new rows inserted go into partition 2 until partition 2 is full
      - Note: with MEMBER CLUSTER, Db2 will insert at the end table space (of each member) first
        - So all the distributed free space reserved up at the front of table space is still free

WAR STORY ABOUT INSERT PROCESSING WITH MEMBER CLUSTER

- What happens next …
  - After the partition 2 is full, “exhaustive” space search kick off for each concurrent insert thread
    - Space map page contention occurs within each data sharing member
    - Since there is so much free space on each data page (PCTFREE at 80%), each data page is visited
    - Page latch only allows one thread to use the page at a time even with LOCKSIZE ROW
    - Because of latch contention on data page, each insert thread keeps searching and continuing to compete for same data pages
    - With multiple inserts per commit scope and LOCKSIZE ROW
      - 1st row is inserted into a data page, but the data page is not secured to avoid it being used by other insert threads
      - 2nd row inserted in the same commit scope, visits the last inserted page, either gets rejected because of page latch contention or the page is already full, and moves on …
    - So the sad story continues with “exhaustive” space search performed across many competing concurrent insert threads

SUMMARY – KEY POINTS

PDUG

- Decide whether the data rows should be clustered or appended at the end
- Sort rows to be inserted into clustering key sequence
- Use table space and index partitioning
- Keep the number of indexes to a minimum and drop low value indexes
- Tune deferred write threshold to drive “trickle write”
- Tune distributed free space
- Use large PRIQTY / SECQTY and large SEGSIZE to reduce frequency of exhaustive space search

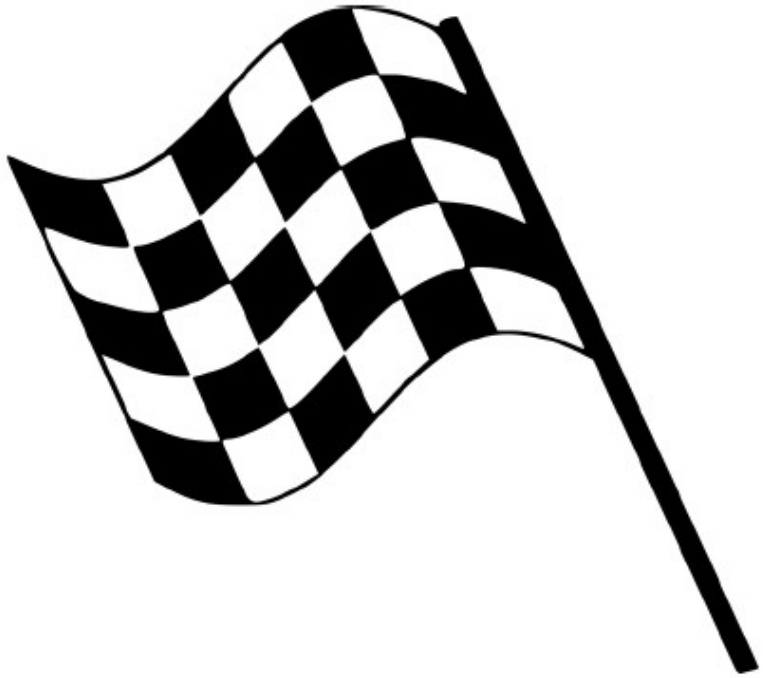
SUMMARY – KEY POINTS …

PDUG

- Use data compression to minimise log record size
- Use faster channel and fast device for active log write throughput
- Use MEMBER CLUSTER and TRACKMOD NO to reduce spacemap page contention and when using LOCKSIZE ROW to reduce data page contention
- Use identity column or Sequence object with caching to efficiently generate a unique key

QUESTIONS

PDUG



INSERT CPU ROUGH RULE OF THUMB

- CPU time estimates based on IBM 9672-Z17 processor model
- Use relative LSPR internal throughput ratio to adjust to other processor models
- Some use cases
  - No index 40 to 80 us
  - One index with no index read I/O 40 to 140us
  - One index with index read I/O 130 to 230us
  - Fives indexes with index read I/O 500 to 800us

INSERT CPU ROUGH RULE OF THUMB …

- General formula CPU time formula

9672-Z17 CPU time = 40 to 80us
+ 30 to 50us * number of indexes
+ 40us * number of I/Os

- Examples
  - If 1 index and no read I/O because of sequential index insert
    - 40 to 80us + 30 to 50us = 70 to 130us
    - CPU cost for write I/O can be ignored because of sequential write of contiguous pages
  - If 3 indexes and 1 random read I/O for each index
    - 40 to 80us + (30 to 50us)*3 + 40us*3*2 (read + write) = 370 to 470us