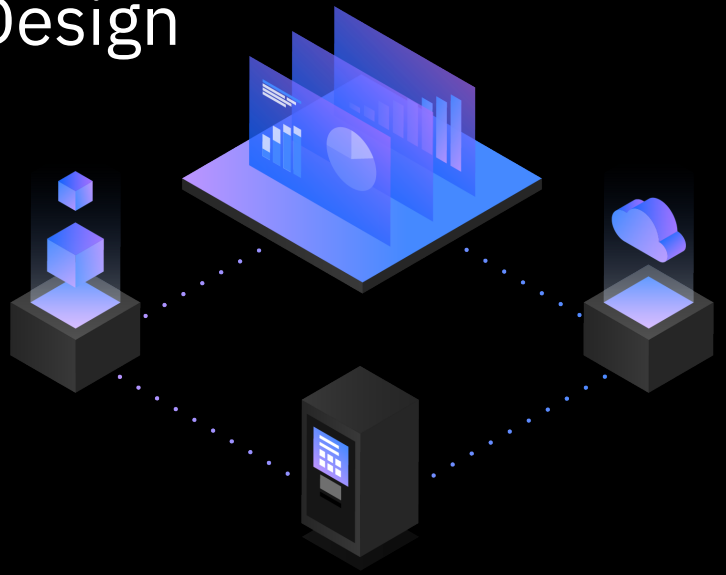
Modern Db2 for z/OS physical Database Design



November 30, 2021

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

Agenda

- Universal table spaces
- Get your partitioning right
- When did you last review your index configuration?
- Thoughts on putting other newer Db2 physical database design-related features to work

Universal table spaces

Your table spaces should be of the universal type

- Why? Because more and more Db2 for z/OS features and functions require the use of universal table spaces:
  - Partition-by-growth
    - Eliminates the 64 GB size limit for table spaces that are not range-partitioned
  - “Currently committed” locking behavior
    - Retrieval of committed data not blocked by inserting, deleting processes
  - Pending DDL
    - Change table, table space and index characteristics via ALTER + online REORG
  - LOB in-lining
    - Store part (or all) of LOB values physically in base table vs. in LOB table space
  - And more (next slide)

More universal-dependent Db2 features

- Continuing from the preceding slide:
  - XML multi-versioning
    - Better concurrency for XML data access, and supports XMLMODIFY function
  - ALTER TABLE with DROP COLUMN
    - An online change, thanks to this being pending DDL
  - Insert partition into middle of range-partitioned table space
  - ALTER COLUMN as pending change versus immediate change
  - Relative page numbering
    - Up to 280 trillion rows, 4000 TB of data in one table
- Absent universal table spaces, you can't use any of these features

Getting to universal table spaces is pretty easy

- How easy? ALTER + online REORG (pending DDL change)
- Because universal table space always holds a single table, getting to universal from single-table non-universal table space is particularly easy:
  - For segmented or simple table space: go to universal partition-by-growth (PBG) with ALTER TABLESPACE with MAXPARTITIONS specification
    - Small MAXPARTITIONS value (even 1) fine for most existing segmented and simple table spaces – can make it larger later (note: default DSSIZE is 4 GB)
  - For classic* partitioned table space: go to universal partition-by-range (PBR) with ALTER TABLESPACE with SEGSIZE specification
    - Go with SEGSIZE 64 (unless number of pages < 128, which is not likely)

* “Classic” = non-universal range-partitioned, using table-controlled vs. index-controlled partitioning

What about multi-table non-universal table spaces?

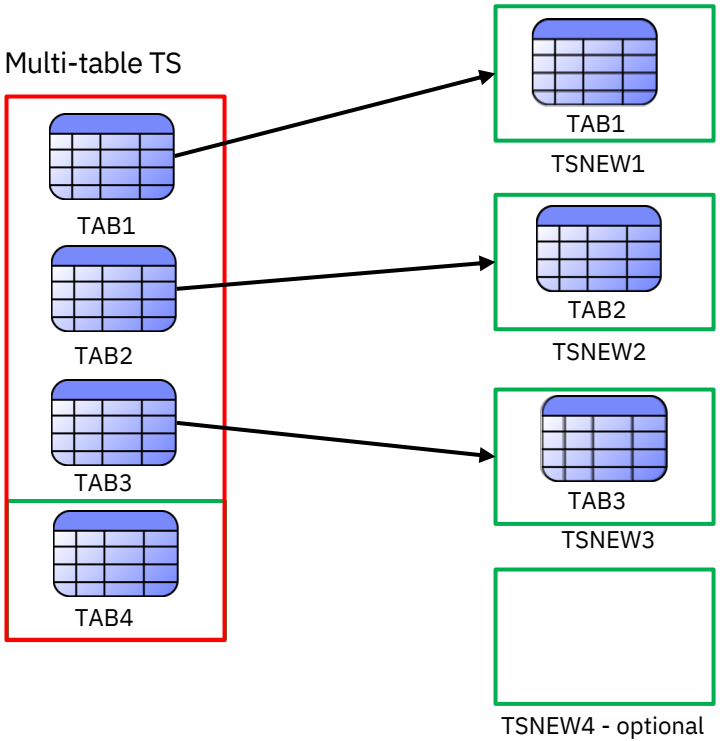
- For years, getting from one multi-table table space to multiple single-table universal table spaces was challenging:
  - Unload data from table in multi-table table space
  - Drop table
  - Re-create table in universal table space
  - Reload data into table

Had to do this for every table you wanted to move from a multi-table table space to a universal table space

- Db2 12 function level 508 (available in October 2020 – APAR is PH29392) provided an online way to get from multi-table table spaces to universal

See next slide

Multi-table table space to UTS PBG – the online way



❶ **`CREATE TABLESPACE TSNEW1 ... MAXPARTITIONS 1`**
`DEFINE NO DSSIZE` ***`appropriate value (e.g. 64G)`***
Repeat this step for every target table space

❷ **`ALTER TABLESPACE`** ***`source table space`*** **`MOVE TABLE TAB1 TO TABLESPACE dbname.TSNEW1`**
- Pending change for source table space
- Repeat this step for every table you want to move
- Requires APPLCOMPAT(V12R1M508) for ALTER

❸ **`Online REORG of source table space`**
- Materializes pending changes – affected tables moved to target table spaces
- Target table spaces are now fully operational

❹ **`Either ALTER TABLESPACE`** ***`source table space`***
`MAXPARTITIONS n` (i.e., leave last table in source TS and ALTER/REORG that TS to UTS PBG)
`or move table to new TS` (as done for previous tables)

More on migration of multi-table table space to UTS PBG

- Target TS must be in same database as source TS, so plan for increase in:
  - OBIDs in database (limit is 32,767)
  - DBD cache in EDM pool
  - Open data sets – DSMAX limit (fix for APAR PH27493 can help here – reduces number of open data sets involved in utility execution)
- MOVE TABLE is a pending DDL change
  - Until materialized, any immediate change and a subset of pending changes prohibited for any table in source TS – even if table not affected by MOVE
  - If necessary, you can DROP PENDING CHANGES
- No PIT recovery for source TS to time before materializing REORG
- Packages dependent on table invalidated when table moved to new TS

Get your partitioning right

Partition-by-range vs. partition-by-growth

- Generally speaking, this debate is relevant for large tables
  - I think of “large” as meaning, “at least 1 million rows” – it would be a little unusual to range-partition a table smaller than that
- Partition-by-growth table spaces are attractive from a DBA labor-saving perspective – they have a “set it and forget it” appeal
  - No worries about identifying a partitioning key and establishing partition ranges, no concern about one partition getting a lot larger than others
  - Just choose reasonable DSSIZE and MAXPARTITIONS values, and you’re done
- That said, I would generally favor range-partitioning a large table

Here’s why (next slide)

Advantages of partition-by-range for large tables

- Maximum partition independence from a utility perspective
  - You can even run LOAD at the partition level for a PBR table space
  - Data-partitioned secondary indexes really maximize partition independence (but, DPSIs not always good for query performance – do predicates reference partitioning key?)
- Enables use of page-range screening by optimizer (limit partitions scanned when predicates reference table's partitioning key)
- Can be a great choice for data arranged by time (see slides 19, 20)
- Maximizes effectiveness of parallel processing (Db2- and user-driven)
- Can use relative page numbering (Db2 12 – more on this to come)

A few more words about PBR vs. PBG

- Ease-of-administration advantage of PBG is real, and PBG can be good choice when data access mainly transactional and most row filtering is at index level
- A word of warning: currently, the only way to change a table space from PBG to PBR is unload/drop/re-create/re-load
  - And, larger a table is, the more advantageous it tends to be to use PBR vs. PBG
  - But, the larger a table is, the more challenging it is to change from PBG to PBR
  - So, really think about PBR vs. PBG for a really large table – you want to get this right, to avoid “buyer’s remorse”

PBR vs. PBG – one more thing

- In making this decision, keep in mind that recent enhancements have made management of PBR table spaces substantially easier than before – for example:
  - Online adjustment of partition limit key values – ALTER and online REORG (Db2 11)
  - Insert new partition into the middle of a PBR table space (Db2 12)
  - Relative page numbering – multiple benefits (Db2 12)

See next slide

Relative page numbering – this is big

- PAGENUM RELATIVE option of ALTER and CREATE TABLESPACE available with Db2 12 function level 500
  - Db2 package through which ALTER or CREATE TABLESPACE with PAGENUM RELATIVE is issued must have APPLCOMPAT value of V12R1M500 or higher
  - What had been thought of as “regular” page numbering now called absolute page numbering
- The difference: instead of every page in table space having a unique number, with RPN page numbering starts over with each partition
  - So, unique identifier of page in an RPN table space is combination of partition number and page number
  - RID length increases for RPN table space: 7 bytes versus 5

What RPN does for you...

- DSSIZE (maximum partition size) can be different for different partitions
  - Formerly, could only specify DSSIZE at table space level
- DSSIZE can be n GB, with n being any integer from 1 through 1024
  - Number of GB formerly had to be a power of 2
  - And, max partition size of 1024 GB – big increase versus former max of 256 GB
- Alter of DSSIZE for partition to a larger value is immediate change – no need to REORG partition in question

What RPN does for you (continued)

- Max number of partitions *no longer affected by DSSIZE or page size*
  - Can have up to 4096 partitions, regardless of DSSIZE or page size
    - Formerly (example): 256 GB DSSIZE and 4 KB page size = 64 partitions, max
  - So, with RPN you can have:
    - Up to 4096 TB of data in one table
    - Up to 280 trillion rows in one table (if using 4 KB page size)

A bit more on RPN table spaces

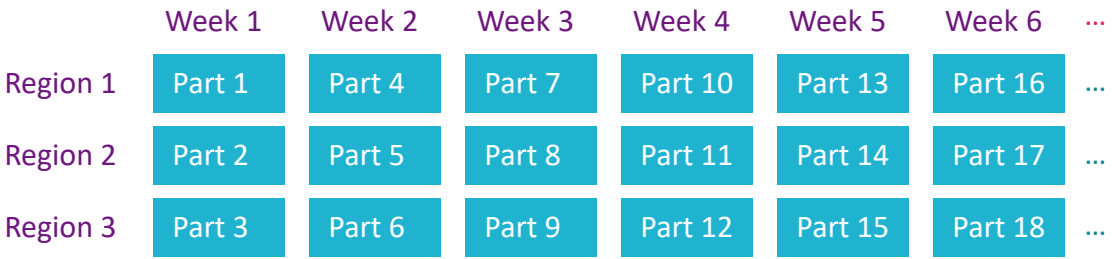
- Convert existing universal PBR table space to RPN via ALTER TABLESPACE with PAGENUM RELATIVE, followed by online REORG of table space
  - Existing classic partitioned table space: ALTER TABLESPACE with SEGSIZE specification, ALTER again with PAGENUM RELATIVE, then REORG table space
- New Db2 12 ZPARM parameter, PAGESET_PAGENUM, specifies default page numbering mechanism to be used for new PBR table spaces
  - Valid values are ABSOLUTE (default) and RELATIVE
  - Can override value of ZPARM when issuing CREATE TABLESPACE
- Partitioned indexes on RPN table spaces get benefits, too, including ability to specify different DSSIZE values for different index partitions

What could 4096 partitions do for you?

- With ALTER TABLE ADD PARTITION, *partitioning by time period* can be an attractive option
  - With 1 week of data per partition, in 10 years you'd only be at 520 partitions
- Would you eventually hit limit on number of partitions for table space, absent ALTER TABLE DROP PARTITION (which we don't yet have)?
  - Yes, but 4096 partitions provides a lot of runway while we wait for that enhancement
  - ALTER TABLE ROTATE PARTITION FIRST TO LAST is an option for keep "rolling" number of time periods in a table
    - Changes mapping of logical to physical partitions, but so does the "insert partition" feature of Db2 12 (more on that to come)

Performance advantages of date-based partitioning

- If more recently inserted rows are the more frequently accessed rows, you've concentrated those in fewer partitions
- Very efficient data purge (and archive) if purge based on age of data – just empty out a to-be-purged partition via LOAD REPLACE with a DD DUMMY input data set (unload first, if archive desired)
- *A second partition key column can give you two-dimensional partitioning* – optimizer can really zero in on target rows
  - Example: table partitioned on ORDER_DATE, REGION



When did you last review your index configuration?

Get rid of indexes that are not doing you any good

- Useless indexes increase the CPU cost of INSERTs, DELETEs, some UPDATEs and many utilities, and waste disk space
- In catalog: use SYSPACKDEP table, and LASTUSED column of SYSINDEXSPACESTATS table, to identify indexes that are not helping the performance of static or dynamic SQL statements, respectively
  - If you find such indexes, do some due diligence, and if they are not needed for something like unique constraint enforcement, DROP THEM
- Also, see if some indexes can be made useless – then drop them
  - Leverage index INCLUDE capability (delivered with Db2 10): if you have unique index IX1 on (C1, C2), and index IX2 on (C1, C2, C3) for index-only access, INCLUDE C3 in IX1 and drop IX2

Index page size: should you go bigger than 4 KB?

- For a long time, 4 KB index pages were your only choice
- Db2 9 made larger index pages – 8 KB, 16 KB, 32 KB – an option
- Larger page sizes are a prerequisite for index compression
- *Some people think large index page sizes are ONLY good for compression enablement – NOT SO*
  - For index with key that is NOT continuously ascending, defined on table that sees a lot of insert activity, larger index page size could lead to major reduction in index page split activity
  - Larger index page size could also reduce number of levels for an index – something that could reduce GETPAGE activity
  - Bigger index pages could also improve performance of index scans

Consider newer index types that can speed queries

- For example, index-on-expression (introduced with Db2 9) could make this predicate stage 1 and indexable:

```
WHERE SUBSTR(C1,4,5) = 'ABCDE'
```

  - The CREATE INDEX statement could look like this:

```
CREATE INDEX SUBSTRIX
ON TABLE T1
(SUBSTR(C1, 4, 5))
USING STOGROUP…
```

- Another example: index on an XML column, to accelerate access to data in XML documents

Thoughts on putting other newer Db2 physical database design-related features to work

Storing data as LOBs even when you don't have to

- If length of some column values > 32,704 bytes, must use LOB data type
- You can use a LOB data type for a column whose values will never exceed 32,704 bytes in length – why would you?
  - Suppose you have a situation in which values in column BIGCOL will average 20,000 bytes in length, and rest of the row will average 200 bytes in length
  - Suppose further that values in the BIGCOL column will rarely be retrieved or referenced in a query predicate
  - If you make BIGCOL a LOB column, its values will be physically placed in an auxiliary table in a LOB table space, and that LOB table space *can have a buffer pool that is different* from the one to which base table's table space is assigned
  - Result: can get LOTS more of the table data that is frequently accessed in the buffer pool, with CPU and elapsed time benefits

LOB in-lining and LOB table space compression

- In-lining: Db2 can store up to first n bytes of LOB value physically in base table – remainder of LOB value (if any) stored in auxiliary table in LOB TS
  - Can significantly improve performance for processes that insert or retrieve LOB values, IF majority of values in column can be completely in-lined in base table
  - Note: even if most values in LOB column *can* be completely in-lined, in-lining could be overall negative for performance if LOB values are rarely accessed
    - Reason: LOB in-lining makes base table rows longer, so GETPAGEs go up and buffer pool hits go down – little offsetting benefit if LOB values rarely accessed
- Compression: with Db2 12, can specify COMPRESS YES for LOB TS
  - Disk and buffer pool space savings (LOB values in compressed form in memory)
  - Uses IBM zEDC technology – z15 boosts performance with on-chip compression

Index compression

- Index compression reduces disk space consumption, period (index pages are compressed on disk, not compressed in memory)
- If you want less disk space usage for indexes, consider index compression
  - CPU overhead of index compression should be fairly low, and you can make it lower by reducing index I/O activity (by assigning indexes to large buffer pools)
    - This is so because much of the cost of index compression is incurred when an index page is read from or written to the disk subsystem
    - Additional cost of I/Os related to index compression is reflected in application class 2 CPU time for synchronous read I/Os, and in CPU consumption of DBM1 address space for prefetch reads, database writes ← 100% zIIP-eligible since Db2 10
    - Best bang for your index compression buck may be realized when compression is used for a relatively small number of your very largest indexes

Reserving space for length-changing UPDATEs

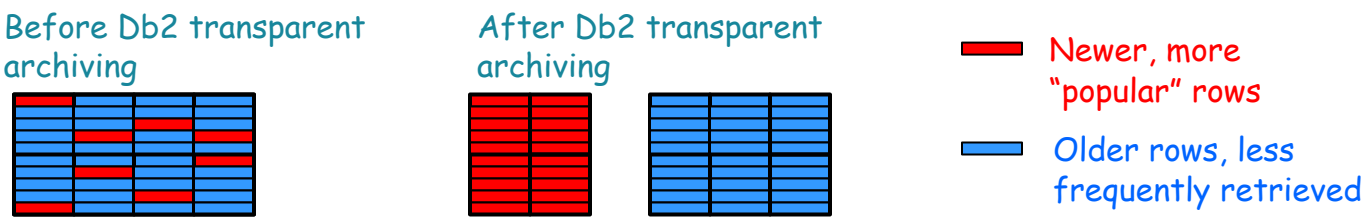
- If row in page X becomes longer because of an UPDATE, and no longer fits in page X, it is moved to page Y and a pointer to page Y is placed in page X
  - That's called an indirect reference, and it's not good for performance
- Db2 11 introduced feature that can reduce indirect references by letting you reserve space in pages to accommodate length-increasing UPDATEs
- PCTFREE *n* FOR UPDATE *m* on ALTER/CREATE TABLESPACE, where *n* and *m* are free space for inserts and updates, respectively
  - PCTFREE_UPD in ZPARM provides default value (PCTFREE_UPD default is 0)
  - PCTFREE_UPD = AUTO (or PCTFREE FOR UPDATE -1): 5% of space in pages will initially be reserved for length-increasing UPDATEs, and that percentage will subsequently be adjusted based on real-time stats

More on PCTFREE FOR UPDATE

- When specified in ALTER TABLESPACE statement, change takes effect next time table space (or partition) is loaded or reorganized
- Good idea to have PCTFREE FOR UPDATE > 0 when a table space gets a lot of update activity and row lengths can change as a result
  - Row-length variability tends to be greatest when nullable VARCHAR column initially contains null value that is later updated to non-null value
  - UPDATESIZE column of SYSTABLESPACESTATS in catalog shows table space growth due to update activity – helps in identifying candidate table spaces
- Goal is fewer indirect references – check on that with these catalog tables:
  - SYSTABLESPACESTATS: REORGNEARINDREF and REORGFARINDREF
  - SYSTABLEPART: NEARINDREF and FARINDREF

Db2 transparent archiving

- Suppose you have a table with 20 years of data, and 95% of requests are for rows inserted within the past 3 months
- Especially if the table is NOT clustered on a continuously-ascending key, over time the newer, “popular” rows will be separated from each other by ever-growing numbers of “old and cold” rows
- With Db2 transparent archiving, ”popular” rows are concentrated in the base table, and “older, colder” rows are stored in associated archive table
  - Can provide significant performance boost for processes accessing “popular” rows



More on Db2 transparent archiving

- Does not complicate query coding – Db2 can make physically separate base and archive tables appear to be single logical table for SELECTs
  - Involves binding program's package with ARCHIVESENSITIVE(YES) and setting built-in Db2 global variable SYSIBMADM.GET_ARCHIVE to 'Y'
  - In that case, for a query that references base table Db2 will execute same query for archive table and will UNION the result sets
- Easy to move row from base table to archive table: just delete row from base table – Db2 will move row to archive table if built-in global variable SYSIBMADM.MOVE_TO_ARCHIVE is set to 'Y' for deleting program
  - 'Y' can be made default value of global variable via a ZPARM, or through use of Db2 profile tables

Implementing Db2 transparent archiving

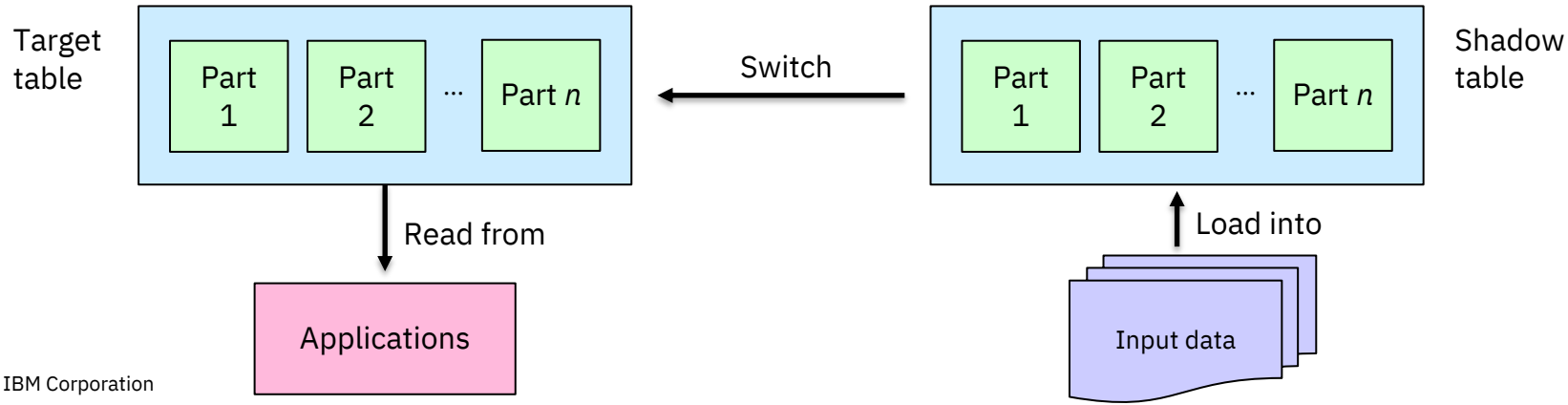
- Easily done: DBA creates table (e.g., T1_AR) to be used as archive for base table T1, and enables archiving via ALTER:

```
ALTER TABLE T1 ENABLE ARCHIVE USE T1_AR;
```

- Note: base table and archive table have to be logically equivalent (same columns, with same names and data types, in same order) but can be different in a physical design sense
  - For example, if base table is in a PBG table space, associated archive table can be range-partitioned
  - Also: no requirement that base and archive have all the same indexes

“Switching out” data in table for other data

- Do-able in non-disruptive with clone tables, but they impose restrictions
- A newer alternative: LOAD REPLACE SHRLEVEL REFERENCE
  - Delivered for Db2 12 via APAR PI69085
  - Uses functionality very similar to that used for online REORG: shadow data sets, which are switched with “original” data sets after LOAD completes
  - ”Original” data available for read while LOAD executing for shadow data sets



Robert Catterall

rfcatter@us.ibm.com