Modern DB2 for z/OS Physical Database Design

TRIDUG

Robert Catterall, IBM

December 9, 2015

Agenda

- Get your partitioning right
- Getting to universal table spaces
- Data security: row permissions and column masks vs. “security views”
- When did you last review your index configuration?
- Thoughts on putting other newer DB2 physical database design-related features to work

Get your partitioning right

First, review existing range-partitioned table spaces

- Do any of them use index-controlled partitioning?
- If “Yes” *then change those to table-controlled partitioning*
- Why? Because it’s easy (more on that in a moment) and it delivers multiple benefits – here are some of my favorites:
    - You can partition based on one key <u>and cluster rows within partitions by a different</u> key (more on this to come)
    - You can have a lot more partitions (up to 4096, vs. 254 for index-controlled)
    - You can do ALTER TABLE *table-name* ADD PARTITION – great when you’re partitioning on something like date
    - There is a <u>non-disruptive</u> path from table-controlled partitioned table space to universal partition-by-range table space (more on this to come)

Identifying index-controlled partitioned table spaces

- It’s easy – submit this query:

```
SELECT TSNAME, DBNAME, IXNAME
FROM SYSIBM.SYSTABLEPART
WHERE IXNAME <> ''
AND PARTITION = 1
ORDER BY 1,2;
```

If the value in the IXNAME column for a table space is NOT blank, the table space uses index-controlled partitioning

This predicate ensures that you’ll get one result set row per table space (remember, SYSTABLEPART contains one row for each <u>partition</u> of a partitioned table space)

Converting to table-controlled partitioning

- Also easy – here is a mechanism I like:
  - Create a data-partitioned secondary index (DPSI) on the table in an index-controlled partitioned table space, specifying DEFER YES so that the index will not be built:

```
CREATE INDEX index-name
ON table-name (column-name) PARTITIONED
DEFER YES
```

  - A DPSI cannot be created on an index-controlled partitioned table space, but instead of responding to the above statement with an error, DB2 will change the table space's partitioning to table-controlled
    - A general rule: if you issue for an index-controlled partitioned table space a DDL statement that is only valid for a table-controlled partitioned table space, DB2 will change to table-controlled partitioning for the table space
  - Complete this process by dropping the just-created DPSI

A note on going to table-controlled partitioning

- For a table-controlled partitioned table space, the partitioning key limit value for the last partition is always strictly enforced
  - Not so for an index-controlled partitioned table space, if the table space was created **without** LARGE or DSSIZE:
    - In that case, if the table is partitioned on date column, and last partition has limit key value of 2015-12-31, a row with a date value of 2016-05-01 will go into last partition with no problem
  - **So**, when an index-controlled partitioned table space created without LARGE or DSSIZE is changed to table-controlled partitioning, the last partition's limit key value will be set to highest possible (if ascending)
    - If that's not what you want (e.g., if you want last partition to have limit key value of 2015-12-31 vs. 9999-12-31):
      - Issue ALTER TABLE with ALTER PARTITION to specify desired partitioning key limit value for last partition
      - If table space has been converted to universal PBR, partition will be placed in AREOR status – new limit will be enforced at next online REORG of partition, and any rows in violation will be placed in discard data set

After changing to table-controlled partitioning…

- Is the formerly partition-controlling index still needed?
  - No longer needed for partitioning – that’s now a table-level thing
  - It no longer has to be the table’s clustering index – alter it with the NOT CLUSTER option *if a different clustering key would be better for the table* (we’re talking here about clustering of rows within partitions)
    - Sometimes a good partitioning key is a lousy clustering key
    - You can ALTER another of the table’s indexes to have the CLUSTER designation, or create a new clustering index for the table
    - **Note:** <u>ALTER INDEX with NOT CLUSTER for partitioning index of index-controlled partitioned table space is another way of getting to table-controlled partitioning</u>
  - If formerly partition-controlling index useless (doesn’t speed up queries, not needed for clustering or uniqueness or referential integrity), DROP IT
    - Save CPU, reclaim disk space

Think: what could 4096 partitions do for you?

- For a traditional table-controlled partitioned or universal partition-by-range table space, you can have up to 4096 partitions, depending on the table space's DSSIZE
  - With ALTER TABLE ADD PARTITION capability, *partitioning by time period* (and by smaller time periods) much more attractive than before
    - For example, think about partitioning by week vs. month – with 10 years of data you'd only be at 520 partitions
  - Now, ALTER TABLE ADD PARTITION is great, but if you want to keep a "rolling" number of time periods in a table's partitions, wouldn't you also want to have ALTER TABLE DROP PARTITION?
    - Yes, but that's a known requirement, and 4096 partitions gives us a lot of runway while we wait for delivery of that enhancement
    - ALTER TABLE ROTATE PARTITION FIRST TO LAST is an option, but it changes the relationship between the number of a partition and the age of the data therein

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)
- Note: *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

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	…
Region 1	Part 1	Part 4	Part 7	Part 10	Part 13	Part 16	…
Region 2	Part 2	Part 5	Part 8	Part 11	Part 14	Part 17	…
Region 3	Part 3	Part 6	Part 9	Part 12	Part 15	Part 18	…

Getting to universal table spaces

Why should your table spaces be universal?

- Most importantly, because a growing list of DB2 for z/OS features and functions require the use of universal table spaces:
  - Partition-by-growth table spaces (DB2 9)
    - Eliminates the 64 GB size limit for table spaces that are not range-partitioned
  - Clone tables (DB2 9)
    - Super-quick, super-easy “switch out” of old data for new in a table
  - Hash-organized tables (DB2 10)
    - Super-efficient row access via an “equals” predicate and a unique key
  - “Currently committed” locking behavior (DB2 10)
    - Retrieve committed data from a table without being blocked by inserting and deleting processes
  - And more (next slide)

More universal-dependent DB2 features

- Continuing from the preceding slide:
  - Pending DDL (DB2 10)
    - <u>Online</u> alteration of table space characteristics such as DSSIZE, SEGSIZE, and page size – via ALTER and online REORG
  - LOB in-lining (DB2 10)
    - Store all or part of smaller LOB values physically in base table versus LOB table space
  - XML multi-versioning (DB2 10)
    - Better concurrency for XML data access, and supports XMLMODIFY function
  - ALTER TABLE with DROP COLUMN (DB2 11)
    - An online change, thanks to this being pending DDL
- Absent universal table spaces, you can't use any of these features – *you may not be using DB2 as effectively as you could*

DB2 10 made getting to universal much easier

- How easy? ALTER + online REORG – that easy (it's a pending DDL change)
- A universal table space can hold a single table, so here are the possibilities for online change to universal from non-universal:
  - Single-table segmented or simple table space to universal partition-by-growth (PBG): ALTER TABLESPACE with MAXPARTITIONS specification
    - A small MAXPARTITIONS value (even 1) should be fine for most of your existing segmented and simple table spaces – you can always make it larger at a later time (and keep in mind that DSSIZE will default to 4 GB)
  - Table-controlled partitioned table space to universal partition-by-range (PBR): ALTER TABLESPACE with SEGSIZE specification
    - Go with SEGSIZE 64 (smaller SEGSIZE OK if table space has fewer than 128 pages, but how many tables that small have you partitioned?)

What about multi-table table spaces?

- For your multi-table segmented (and simple) table spaces, there is not a direct path to the universal type
- That leaves you with a couple of choices:

   A. Go from n tables in one table space to 1 table in n universal PBG table spaces via UNLOAD/DROP/re-CREATE/re-LOAD

   - Could be feasible for a table space that holds just a few not-so-big tables

   B. Wait and see if a path from multi-table table space to universal is provided in a future release of DB2 (it’s a known requirement)

   - Very understandable choice, especially if you have (as some do) a segmented table space that holds hundreds (or even thousands) of tables
   - If a table space does hold hundreds or thousands of tables, you could do unload/drop/re-create/re-load for an individual table if a universal-only DB2 feature would be particularly valuable for that table

A couple more things…

1. Converting a table space to universal via ALTER and online REORG will invalidate packages that have a dependency on the table space
   - To identify these packages ahead of time, query SYSPACKDEP catalog table
2. You may be thinking, “I don’t want to convert my smaller segmented and simple table spaces to universal partition-by-growth, because partitioning is for BIG tables”
   - Don’t get hung up on the “partition” part of partition-by-growth – unless the size of the table reaches the DSSIZE (which defaults to 4 GB), it will never grow beyond 1 partition
   - Bottom line: a segmented table space that holds 1 relatively small table will be, when converted to universal, a relatively small PBG table space
     - In other words, *PBG is appropriate for small, as well as large, table spaces*

Data security: row permissions and column masks vs. “security views”

Background

- A long-standing requirement: restrict access to certain rows in a table, and/or to mask values in certain columns, based on user's role
- This need was typically addressed through the use of "security views," which have some shortcomings:
  - For one thing, a view can't have the same fully qualified name as the underlying table – that makes job harder for developers
    - If unqualified name is the same (allowing use of 1 set of unqualified SQL statements for a program), you need different packages and collections for different qualifiers, and you have to use the right collection at run time
    - If unqualified view name is different from table name, need multiple sets of SQL statements for a program
  - On top of that, security views can really proliferate (including views on views) – that makes job harder for DBAs

DB2 10 gave us a new way to address this need

- Row permissions and column masks
  - A row permission provides predicates that will filter out rows for a given table when it is accessed by a specified individual authorization ID, or RACF (or equivalent) group ID, or DB2 role
  - A column mask functions in a similar way, but it provides a case expression that masks values of a column in a table
  - The predicates of a row permission and the case expression of a column mask are automatically added by DB2 to any query – static or dynamic – that references the table named in the permission or mask
- The really good news: *with row permissions and column masks, all SQL references the base table* – there is no need to reference a differently-named view
  - Job of data security effectively separated from job of programming

The flip side, for row permissions

- Planning is important!
  - Consider this row permission:

```
CREATE PERMISSION SECHEAD.ROW_EMP_RULES ON SPF.EMP
FOR ROWS
WHERE (VERIFY_GROUP_FOR_USER(SESSION_USER, 'MANAGER') = 1
AND WORKDEPT IN('D21', 'E21'))
ENFORCED FOR ALL ACCESS;
```

  - Suppose this permission is enabled, and row access control is activated for the table, and no other permissions exist for the table – what then?
    - People with the group ID ‘MANAGER’ can retrieve rows for departments D21 and E21, <u>and no one else will be able to retrieve ANY data from the table</u>
  - So, THINK about this, and have ALL the row permissions you’ll need for a table (more restrictive and less restrictive) set up before you activate row access control for the table

Is there still a place for security views?

- For controlling data access at the row level, I would say that row permissions are generally preferable to views, *provided you have done the proper advance planning* (see preceding slide)
- For controlling access at the column level, I can see where you still might want to use a view:
  - If you want to mask quite a few columns in a table
    - Do-able with <u>one</u> view, whereas you need a column mask <u>per</u> column that you're masking
  - If you want to make it appear that a column doesn't even exist in a table
    - If column isn't in view's select-list, it's essentially invisible
    - With a column mask, user sees that the column is there, but receives masked data values

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 and DELETEs (and some UPDATEs) and many utilities, and waste disk space
- Use SYSPACKDEP catalog table, and the LASTUSED column of SYSINDEXSPACESTATS, to identify indexes that are not helping the performance of static and 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, <u>DROP THEM</u>
- Also, see if some indexes can be <u>made</u> useless – then drop them
  - As previously mentioned, change to table-controlled partitioning can make formerly partition-controlling index useless (slide 8)
  - Leverage index INCLUDE capability introduced with DB2 10:
    - If you have unique index IX1 on (C1, C2), and index IX2 on (C1, C2, C3, C4) for index-only access, INCLUDE C3 and C4 in IX1 <u>and drop IX2</u>

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 an index with a key that is NOT continuously ascending, defined on a table that sees a lot of insert activity, a larger index page size could lead to a 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 for operations involving index scans

Consider new index types that could speed queries

- For example, index-on-expression, which could make the following predicate stage 1 and indexable:

```
WHERE SUBSTR(COL1,4,5) = ‘ABCDE’
```

- Another example: index on an XML column, to accelerate access to XML data in a DB2 table

Thoughts on putting other newer DB2 physical database design-related features to work

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

- Generally speaking, this debate is relevant for a large table
  - Range partitioning a little unusual for a table with fewer than 1 million rows
- Partition-by-growth table spaces are attractive from a DBA labor-saving perspective – they have kind of 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, my preference would usually be to range-partition a table that holds millions (or billions) of rows

Here’s why (next slide)

Advantages of partition-by-range for BIG tables

- Maximum partition independence from a utility perspective
  - You can even run LOAD at the partition level for a PBR table space
  - You can have data-partitioned secondary indexes, which maximize partition independence in a utility context (note: DPSIs not always good for query performance – do predicates reference partitioning key?)
- Enables the use of page-range screening (limit partitions scanned when predicates reference table’s partitioning key)
- Can be a great choice for data arranged by time (see slides 9, 10)
- Can maximize effectiveness of parallel processing, especially for joins of tables partitioned on same key
- Still, ease-of-administration advantage of PBG is real, and PBG can be a very good choice when data access is predominantly transactional and most row filtering is at index level

Hash organization of data

- Really is a niche solution – good for performance if data access is:
  - Dominated by singleton SELECTs that reference not just a unique key, but one particular unique key (the table's hash key) in an "equals" predicate

    -and-
  - Truly <u>random</u>
    - If data is accessed via singleton SELECTs in a "do loop" in a batch job, and predicate values come from a file that is sorted by target table's clustering key, cluster-organized data may well out-perform hash-organized data
    - The reason: dynamic sequential prefetch

LOB in-lining

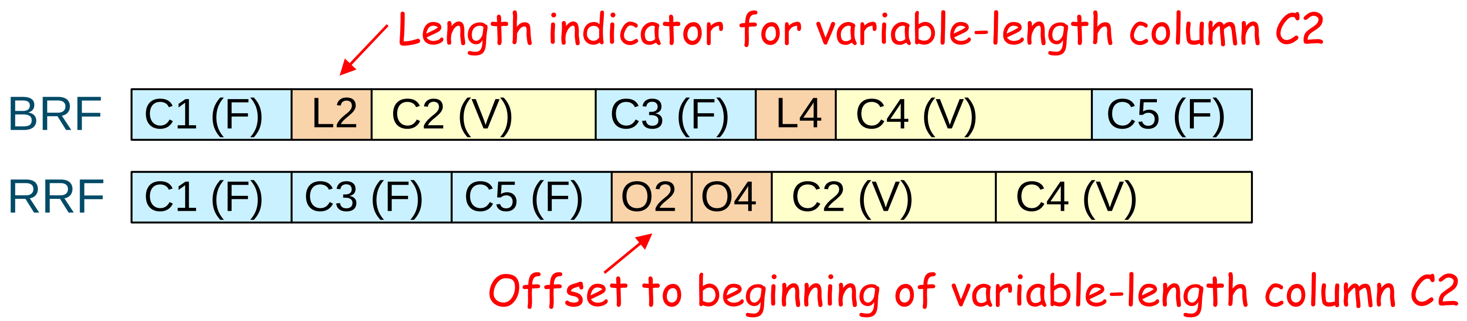
- In-lining a LOB column is almost certainly NOT good for performance if a majority of the values in the column can’t be completely in-lined
  - Exception: for CLOB column, if an index-on-expression, built using SUBSTR function applied to in-lined part of the column, is valuable, you might inline, even if most values cannot be completely in-lined
- Even if most values in a LOB column could be completely in-lined, in-lining those LOBs could be bad for performance <u>if the LOBs are rarely accessed</u>
  - In that case, you’ve made base table rows longer, which could increase GETPAGE activity and reduce buffer pool hits, with little offsetting benefit because the LOBs are not often retrieved by programs

Index compression

- Index compression reduces disk space consumption, period (index pages are compressed on disk, not compressed in memory)
- If you want to reduce the amount of disk space occupied by indexes, go ahead and compress
  - The CPU overhead of index compression should be pretty low, and you can make it lower by reducing index I/O activity (typically done by assigning indexes to large buffer pools)
    - This is so because the cost of index compression is incurred when an index page is read from or written to the disk subsystem – not when an index page is accessed in memory
    - CPU cost of compression is reflected in an application’s class 2 (i.e., in-DB2) CPU time for synchronous read I/Os, and in the CPU consumption of the DB2 DBM1 address space for $\underbrace{\text{prefetch reads and database writes}}_{\text{100\% zIIP-eligible starting with DB2 10}}$

Reordered row format (RRF)

- Getting table spaces into RRF vs. BRF (basic row format) doesn't have to be at the very top of your to-do list, but get it done
- If value of RRF parameter in ZPARM is set to ENABLE (the default):
  - New table spaces (and new partitions added to existing partitioned table spaces) will automatically use reordered row format
  - An existing BRF table space (or partition of same) will be converted to RRF via REORG or LOAD REPLACE of the table space (or partition)
- Benefits: 1) more efficient navigation to variable-length columns in a row, and 2) you're positioning yourself for the future


Length indicator for variable-length column C2
BRF
C1 (F)
L2
C2 (V)
C3 (F)
L4
C4 (V)
C5 (F)
RRF
C1 (F)
C3 (F)
C5 (F)
O2
O4
C2 (V)
C4 (V)
Offset to beginning of variable-length column C2

Reserving space for length-changing UPDATEs

- If a 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 helps to avoid indirect references by allowing you to 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

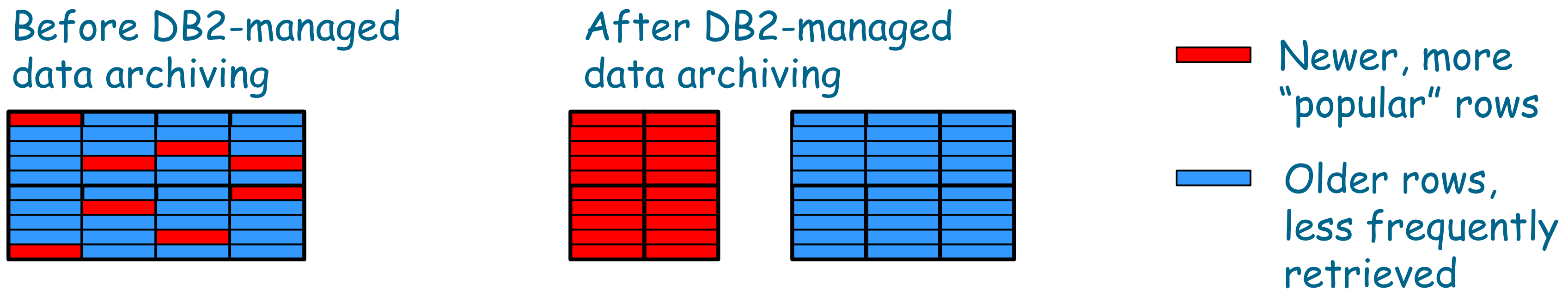
- When specified in ALTER TABLESPACE statement, change takes effect next time table space (or partition) is loaded or reorganized
- Good idea to have PCTFREE FOR UPDATE > 0 when a table space gets a lot of update activity and row lengths can change as a result
  - And remember, a compressed row’s length can change with an update, even if the row’s columns are all fixed-length
  - Row-length variability tends to be greatest when nullable VARCHAR column initially contains null value that is later updated to non-null value
  - New UPDATESIZE column of SYSTABLESPACESTATS catalog table shows how a table space is growing (or not) due to update activity
- If PCTFREE FOR UPDATE > 0, should be fewer indirect references
  - SYSTABLESPACESTATS: REORGNEARINDREF and REORGFARINDREF
  - SYSTABLEPART: NEARINDREF and FARINDREF

Clone tables – not well understood by some folks

- Clone table functionality lets you quickly, programmatically, non-disruptively "switch out" data in a table
- Suppose you have a situation in which an application accesses data in table X, which contains sales data from the prior quarter
  - At the end of Qn, you want the table to contain data for Qn, not Qn-1, because Qn has just become the "previous" quarter
  - Could use LOAD REPLACE to replace Qn-1 data in table X with Qn data
    - ☹ Takes too long, table unavailable during LOAD, using utility kind of clunky
  - Could load Qn data into table Y, then RENAME TABLE X to Z, Y to X
    - ☹ RENAME TABLE causes packages that reference table X to be invalidated
  - Better: alter table X to add clone, load Qn data into clone, EXCHANGE DATA so references to table X resolve to what had been the clone table
    - ☺ Very quick (just need drain on table X), no utilities, no package invalidations

DB2-managed data archiving

- NOT the same thing as system time temporal data
  - When versioning (system time) is activated for a table, the “before” images of rows made “non-current” by update or delete are inserted into an associated history table
  - With DB2-managed archiving, rows in archive table are current in terms of validity – they’re just older than rows in the associated base table
    - When most queries access rows recently inserted into a table, moving older rows to archive table *can improve the performance of newer-row retrieval*
    - Particularly helpful when data clustered by non-continuously-ascending key
    - People have long done this themselves – DB2 11 makes it easier


Before DB2-managed data archiving
After DB2-managed data archiving
Newer, more “popular” rows
Older rows, less frequently retrieved

DB2-managed data archiving – how it's done

1. DBA creates table (e.g., T1_AR) to be used as archive for table T1
2. DBA tells DB2 to enable archiving for T1, using archive table T1_AR

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

3. Program deletes to-be-archived rows from T1

   - If program sets built-in global variable SYSIBMADM.MOVE_TO_ARCHIVE to 'Y', *all it has to do is delete from T1* – DB2 will move deleted rows to T1_AR

4. Bind packages appropriately (affects static and dynamic SQL)

   - If a program is to ALWAYS access ONLY the base table, it should be bound with ARCHIVESENSITIVE(NO)
   - If a program is to SOMETIMES or ALWAYS access rows in the base table and the archive table, it should be bound with ARCHIVESENSITIVE(YES)
     - If program sets built-in global variable SYSIBMADM.GET_ARCHIVE to 'Y', and issues SELECT against base table, DB2 will automatically drive that SELECT against associated archive table, too, and will merge results with UNION ALL

"That's all, folks! Make sure that your physical database design reflects <u>today's</u> DB2, not 1990s DB2!"

Thank You