Modern Db2 for z/OS Physical Database Design

New England Db2 Users Group
June 20, 2019
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 and cluster rows within partitions by a different 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 non-disruptive 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 partition 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
    ```

  - DPSI cannot be created on index-controlled partitioned table space, but instead of responding to above statement with an error, Db2 will change 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 table-controlled partitioned table space, partitioning key limit value for 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 2018-12-31, row with a date value of 2019-03-01 would go into last partition with no problem
  - So, when index-controlled partitioned table space created without LARGE or DSSIZE is changed to table-controlled partitioning, 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 2018-12-31 vs. 9999-12-31), issue ALTER TABLE with ALTER PARTITION to specify desired partitioning key limit value for last partition
    - If table space converted to universal PBR, partition placed in AREOR status – new limit will be enforced at next online REORG of partition (any rows in violation 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:** ALTER INDEX with NOT CLUSTER for partitioning index of index-controlled partitioned table space is another way of getting to table-controlled partitioning
  - 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 traditional table-controlled partitioned or universal partition-by-range table space, you can have up to 4096 partitions, depending on table space’s DSSIZE
  - With ALTER TABLE ADD PARTITION capability, \textit{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?
    * We don’t have that capability yet (known requirement), but 4096 partitions gives you a lot of runway while you wait for delivery of that enhancement
    * ALTER TABLE ROTATE PARTITION FIRST TO LAST is an option – it changes mapping of logical to physical partitions (as does Db2 12 “insert partition” feature – 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).

- 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

| 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
  - “Currently committed” locking behavior (Db2 10)
    - Retrieve committed data from table without being blocked by inserting and deleting processes
  - Pending DDL (Db2 10)
    - Online 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
  - And more (next slide)
More universal-dependent Db2 features

- Continuing from the preceding slide:
  - 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
  - Insert partition into middle of range-partitioned table space (Db2 12)
  - ALTER COLUMN as pending versus immediate change (Db2 12)
  - Relative page numbering (Db2 12)
    - Up to 280 trillion rows, 4000 TB of data in one table
  - “Fast insert algorithm” (Db2 12)
    - Substantial increase in insert throughput, improved CPU efficiency

- 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)

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 for path from multi-table table space to universal to be provided in a future Db2 12 function level (it’s a known requirement)

  – Option B is an 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 smaller segmented and simple table spaces to universal partition-by-growth, because partitioning is for BIG tables”
   - Don’t get hung up on “partition” part of partition-by-growth – unless size of table reaches 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

- Need typically addressed through use of “security views,” which have shortcomings:
  - For one thing, view can’t have same fully qualified name as underlying table – that makes job harder for developers
    - If unqualified name is same (allowing use of 1 set of unqualified SQL statements for program), you need different packages and collections for different qualifiers, and you have to use 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
An alternative way to address this need

- **Row permissions and column masks (introduced with Db2 10)**
  - Row permission provides predicates that will filter out rows for a given table when it is accessed by specified individual authorization ID, or RACF 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
  - Predicates of row permission and case expression of column mask are automatically added by Db2 to any query – static or dynamic – that references table named in permission or mask (true for all IDs – even “super-users” such as SYSADMs)

- **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:

    ```sql
    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, and no one else will be able to retrieve ANY data from the table
    - 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 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 one view, whereas you need a column mask per 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 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, DROP THEM.

- Also, see if some indexes can be made 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 (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 for operations involving index scans
Consider newer index types that could speed queries

- For example, index-on-expression, which could make 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
  - 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 (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

- Can use relative page numbering (Db2 12 – more on this to come)

- Still, ease-of-administration advantage of PBG is real, and PBG can be good choice when data access predominantly transactional and most row filtering at index level
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 if the LOBs are rarely accessed
  - 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 amount of disk space occupied by indexes, go ahead and compress
  - CPU overhead of index compression should be pretty low (especially if insert and/or update rate is not too high), 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 (i.e., in-Db2) 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
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

<table>
<thead>
<tr>
<th>BRF</th>
<th>C1 (F)</th>
<th>L2</th>
<th>C2 (V)</th>
<th>C3 (F)</th>
<th>L4</th>
<th>C4 (V)</th>
<th>C5 (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRF</td>
<td>C1 (F)</td>
<td>C3 (F)</td>
<td>C5 (F)</td>
<td>O2</td>
<td>O4</td>
<td>C2 (V)</td>
<td>C4 (V)</td>
</tr>
</tbody>
</table>

Length indicator for variable-length column C2
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 introduced feature that can reduce 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.
  - Remember: update of compressed row can change length, even if all columns 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.
Db2-managed data archiving (aka “transparent archiving”)

- NOT same thing as system time temporal data
  - When versioning (system time) is activated for a table, “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 associated base table
    - When most queries access rows recently inserted into a table, moving older rows to archive table can improve performance of newer-row retrieval
    - Particularly helpful when data clustered by non-continuously-ascending key
    - People have long done this themselves – transparent archiving makes it easier
Db2-managed data archiving – how it’s done

- DBA creates table (e.g., T1_AR) to be used as archive for table T1
- DBA tells Db2 to enable archiving for T1, using archive table T1_AR
  - ALTER TABLE T1 ENABLE ARCHIVE USE T1_AR;
- Program moves to-be-archived rows from T1 to T1_AR by way of DELETE
  - If built-in global variable SYSIBMADM.MOVE_TO_ARCHIVE is set to ‘Y’, all program has to do is delete from T1 – Db2 will move deleted rows to T1_AR
- Bind packages appropriately (affects static and dynamic SQL)
  - If program will ALWAYS access ONLY base table, bind with ARCHIVESENSITIVE(NO)
  - If program will SOMETIMES or ALWAYS access rows in base table and archive table, it should be bound with ARCHIVESENSITIVE(YES)
    - If built-in global variable SYSIBMADM.GET_ARCHIVE set to ‘Y’, and program, issues SELECT against base table, Db2 will automatically drive that SELECT against associated archive table, too, and will merge results with UNION ALL
Need a way to quickly “switch out” data in table for other data?

- Clone tables provided way to do that, but impose multiple restrictions
- A newer alternative that might be better for you: online LOAD REPLACE
  - Official terminology: LOAD REPLACE SHRLEVEL REFERENCE
  - Uses functionality very similar to that used for online REORG: shadow objects
  - APARs: PI67793, PI74641 for Db2 11; PI69095 for Db2 12)
Maximizing insert throughput

- Table space defined with MEMBER CLUSTER
  - Tells Db2 to ignore clustering when inserting into table (clustering can be restored later, if desired, via REORG)
  - Table defined with APPEND YES could potentially improve insert throughput still further (send all rows to end of table)

- New with Db2 12: insert algorithm 2 (also known as “fast insert algorithm”)
  - Valid only for universal table space defined with MEMBER CLUSTER
    - Table can be defined with APPEND YES, but that is not a requirement
  - Can substantially improve insert throughput, and significantly improve CPU efficiency for inserting processes

- Specify for CREATE or ALTER TABLESPACE
How fast insert algorithm works

- **V11 behavior**
  - Search for space for newly-inserted rows can impede throughput

- **Db2 V12R1M500 (or later) implementation**
  - In-memory structure contains pages which are currently available for insert
  - Background task puts pages with space for new rows into pipe – inserting processes get pages from pipe
Relative page numbering (Db2 12 – function level M500)

- Instead of table space pages being numbered in continuously-ascending fashion from beginning to end, with RPN page numbering starts again with each partition
  - So, unique identification of a page in an RPN table space is combination of partition number and page number
  - RID length goes from 5 to 7 bytes

- Restraints lifted:
  - DSSIZE can be different for different partitions
  - DSSIZE can be nG (G=GB), with n being any integer from 1 through 1024
  - Alter of DSSIZE to larger value is immediate change – no REORG necessary
  - Number of partitions no longer dependent on DSSIZE
    - Can have up to 4096 partitions, each of which can be up to 1024 GB in size (up to 4 PB of data)
    - Up to 280 trillion rows in one table (if 4KB pages used)
A little more on RPN table spaces

- Valid for universal partition-by-range table space

- Existing PBR table space that uses absolute page numbering: online REORG of entire table space needed to change to RPN
  - If existing table space is table-controlled range-partitioned but not universal (i.e., if a “classic” partitioned table space):
    - Alter once with a SEGSIZE to get it to universal PBR
    - Alter a second time to specify PAGENUM RELATIVE
    - One online REORG after those two ALTERs changes table to universal RPN

- Partitioned indexes on RPN table spaces get similar benefits, including ability to specify different DSSIZE values for different index partitions
Robert Catterall
rfcatter@us.ibm.com