17679: Best Practices for Efficient DB2 Physical Index Design

Hal Steiner
Bank of America / Merrill Lynch

Wednesday, August 12, 2015: 11:15 AM-12:30 PM
Southern Hemisphere 4
Walt Disney World Dolphin

Presentation Outline

- In this presentation, Hal takes a deep dive technical approach to modern day, efficient, physical DB2 Index design.
- He will review the differences in index types: clustered and non-clustered, partitioned and non-partitioned, UNIQUE and non-UNIQUE.
- He will explain the costs and benefits of important index features such as Index Compression, the effect of PADDED vs. NOT PADDED, NPSI vs. DPSI, PIECESIZE, and others, with a focus on best practices.
- Hal will also offer some modern ideas about Index Page size and index buffer pools.
- In particular, Hal will look at partitioned indexes contrasting classic vs. UTS/PBR and make several recommendations, sharing benchmark results from his own experience.
- The session provides several opportunities for Q & A plus general comments from the audience.

Knowledge Level Setting Quiz

You can assume DB2 V10 NFM to answer these questions:

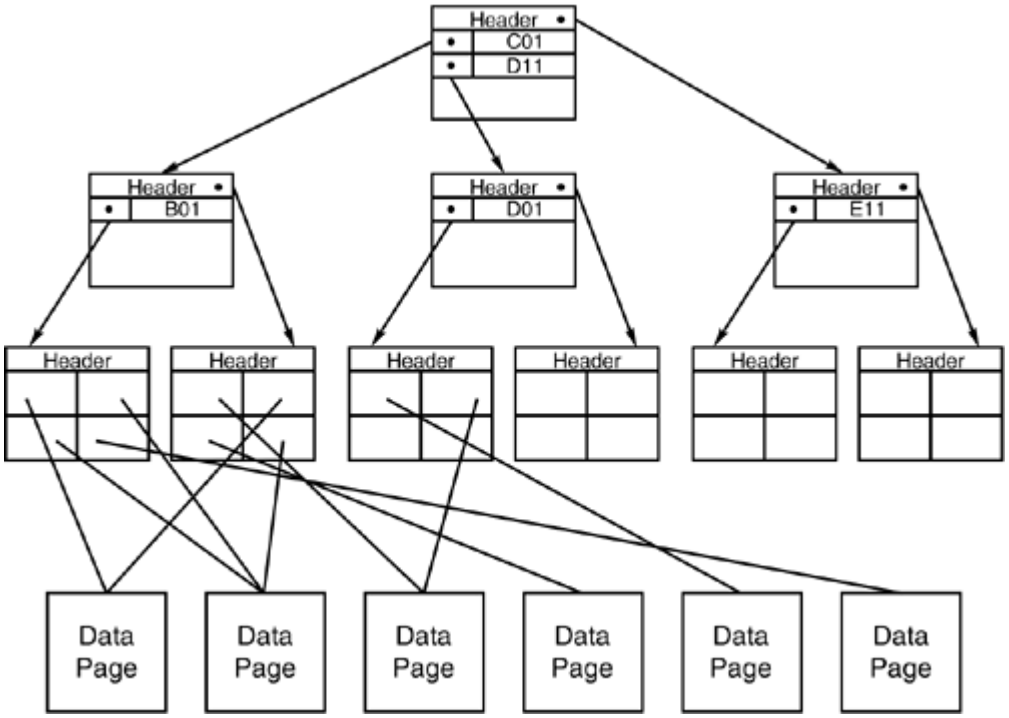
1) What is the maximum size in bytes of a composite Index Key?
   A) 255 B) 512 C) 4096 D) 2000

2) What is the maximum size NPI index?
   A) 64GB B) 2TB C) 128 TB D) unlimited

3) True or False:

   A partitioning Index needs to be clustered as well.
   A DPSI (Data Partition Secondary Index) can't be UNIQUE.
   Index Splits are always 50-50% with half the entries copied to a new index page.
   To convert a Classic Partitioned tablespace to a PBR you need to DROP and CREATE.
   Index and Data compression both share the same Compression Dictionary to save space.

Typical DB2 Index Structure (Not Clustered)



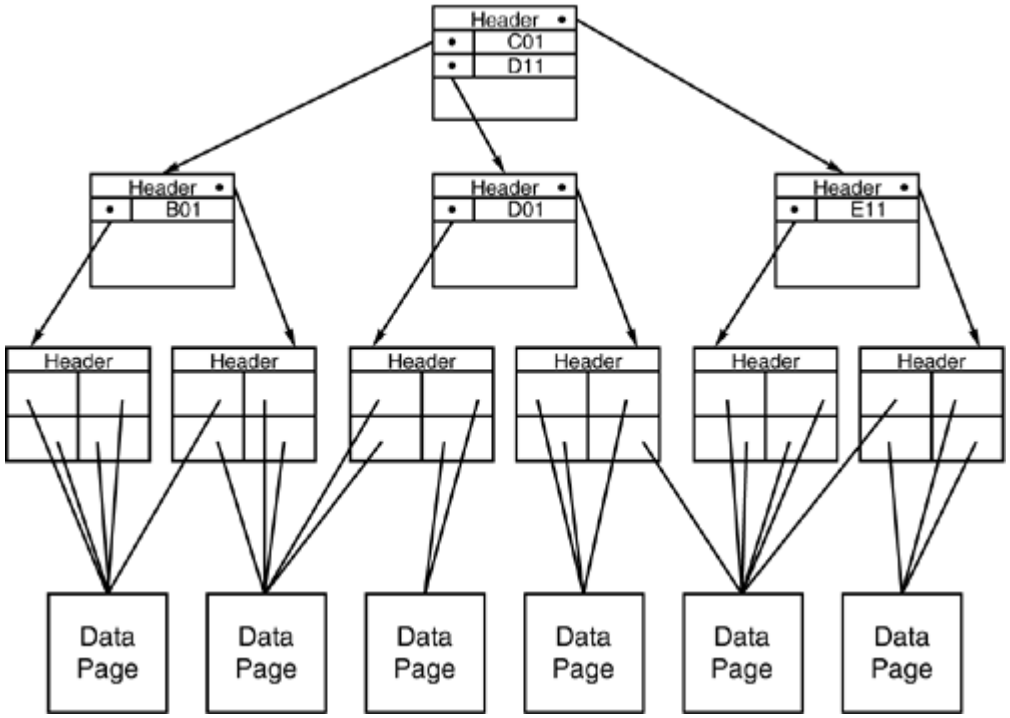
Root Page

Non Leaf

Leaf Pages

Note: Fan-out is determined by both physical index page size, and key length

Typical DB2 Index Structure (Clustered!)



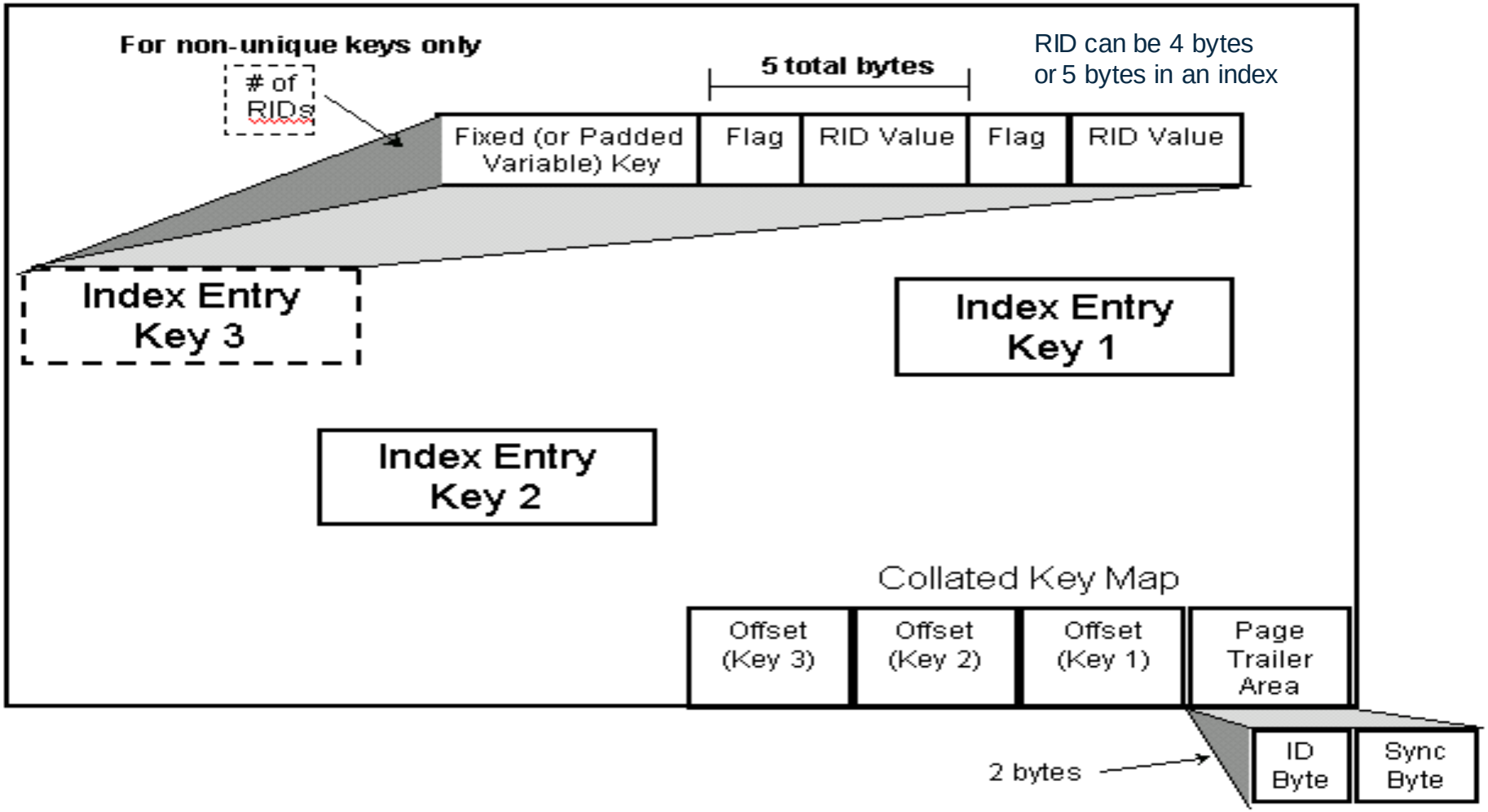
Root Page

Non Leaf

Leaf Pages

The Index defined with CLUSTER determines the physical order of the rows in the data table, resulting in a minimum # of GETPAGEs when SELECTing a set of data rows. The CLUSTER option “matters” to INSERT and REORG and the Optimizer.

Index Page Layout



Index Compression – NOT PADDED option

- Starting in DB2 V8, we had the option of not padding VARCHAR column data which appears in DB2 indexes with spaces. Prior to V8, and still in some places by default, an index on a column like LASTNAME VARCHAR(40), would contain SMITH and 35 spaces.

- BEST PRACTICE: Specify NOT PADDED for any index containing VARCHAR columns, assuming there is some fluctuation in the length of the data. You can set the installation default for PAD INDEXES BY DEFAULT in the installation panel DSNTIPE – many haven't done this.

- This will definitely cause even UNIQUE index entries to become variable length, but you'll increase the potential fan-out of each index page. Otherwise there is a stiff penalty to PADDED indexes in a reduced fan-out.

- There were some negative user experiences in V8 with NOT PADDED indexes; however, with V9 and later this concern seems to have disappeared.

Index Compression – COMPRESS option

- Starting in DB2 V9, we were given the option of compressing Indexes in the CREATE INDEX statement (the default is still COMPRESS NO).
- This is not the Lempel-Ziv algorithm used by Data Compression. Data Compression and Index Compression are two entirely different algorithms and have to be considered independently.
- Index Compression, when turned on selectively, can reduce the DASD footprint of certain large DB2 Indexes by 25-75%.
- Index Compression should be a "wash" in terms of CPU overhead for most tables due to the fact that index compression and expansion is done infrequently and asynchronously.
- With data compression, rows are compressed on each INSERT or UPDATE and expanded for SELECT each and every data GETPAGE. Data Compression usually has the beneficial effect of putting more rows into the same size physical page increasing buffer hit ratios. And of course the physical table size is reduced, meaning fewer I/O operations needed for certain multi-row multi-page operations, such as scans.
- Bottom line: This can be a major DASD save for large indexes without having any appreciable impact on any other application costs or timings.

Data vs. Index Compression

FUNCTION	Data	Index
Level	Row	Page
Compressed Disk	Yes	Yes
Compressed in BP	Yes	No
Compressed in Log	Yes	No
Compressed in IC	Yes	No
Compression Dictionary	Yes	No
Hardware Assisted?	Yes, CMPSC	Partial
Buffer Pool Sizes	4K, 8K, 16K, 32K	8K, 16K, 32K
Page Size on Disk	= buffer size	Always 4K
Average Compress Ratio	10-90%	25-75%

Index Compression – COMPRESS option

- DB2 V9 Index Compression is *NOT* dictionary based. It has three components to the algorithm.

  A. Prefix compression, with Front Keys examined and full or partial reoccurring values suppressed. So if your leading column index keys (also known as key prefixes) tend to hold common values, you will benefit. For example, an Index Key of STATE, CITY, STREET ADDRESS would be a good candidate for key compression, since DB2 wouldn't store redundant prefix values.

  B. RID list compression. When a given NONUNIQUE index has many duplicates, several of these rows might exist in the same page in the table. Without compression, entire RIDs (4 byte or 5 byte) are always stored. With index compression, page numbers are stored only once in the index entry with the single byte row numbers chained until the next page number. This is particularly effective with NONUNIQUE Clustered indexes, and other low cardinality INDEXes which have a good CLUSTERRATIO

  C. In-memory-only key maps. An uncompressed index entry leaf page contains a key map, which contains a two byte pointer entry for every distinct key value stored in the page. If the index is compressed, this map will not be stored on disk (it will be reconstructed, at relatively low cost, when the leaf page is read into memory). This compression technique nicely complements the RID list compression mechanism, as it is most effective for high-cardinality indexes (especially those with short keys, as the more distinct key values a page holds, the more space the key map occupies).

Index Compression – COMPRESS option

- With DB2 index compression, compressed 4K physical disk pages are expanded into an 8K, 16K, or 32K buffer pool page *ONLY* once when READ from Disk. In other words, you only pay at I/O time, not on every GETPAGE. Index Prefetch performs the index expansion ASYNCHRONOUSLY. With COMPRESS YES, you will need to assign the index to a DB2 BUFFER POOL which is > 4K, such as BP8K3.

- Unlike data compression, with index compression, a buffer hit does not require any compression/decompression. On output, when index records are updated, index pages are compressed by the deferred write engine. Starting with DB2 10, DB2 prefetch read and database write operations became 100% zIIP-eligible.

- Who pays the bill? If the I/O is of the prefetch read variety, or an index write, the leaf page compression cost will be charged to the DB2 database services address space (aka DBM1). If it's a synchronous read I/O, index compression overhead will affect the class 2 CPU time of the application process for which the on-demand read is being performed.

- BOTTOM LINE: CPU Cost of Index Compression is mostly inconsequential in the aggregate, and at most should be in the single digits, percentagewise. Some operations will be slower: Synchronous READ, and the DB2 REBUILD utility for example. Some will be faster: such as index scans, jobs which do heavy inserts by the reduction of frequency of CI splits.

- There is a utility DSN1COMP which will guesstimate the benefit of Index Compression. See example on next page.

Index Compression – COMPRESS option

- Output might look like this – which size Buffer Index Page will you want?

```
.          8  K Page Buffer Size yields a
.         51  % Reduction in Index Leaf Page Space
.            The Resulting Index would have approximately          8K
.         49  % of the original index's Leaf Page Space
.            No Bufferpool Space would be unused
-------------------------------------------------
.         16  K Page Buffer Size yields a
.         75  % Reduction in Index Leaf Page Space
.            The Resulting Index would have approximately         16K
.         25  % of the original index's Leaf Page Space
.            No Bufferpool Space would be unused
-------------------------------------------------
.         32  K Page Buffer Size yields a
.         75  % Reduction in Index Leaf Page Space
.            The Resulting Index would have approximately         32K
.         25  % of the original index's Leaf Page Space
.         50  % of Bufferpool Space would be unused to ensure keys fit into compressed buffers
```

- Recommendation: The right index page size will be the one that maximizes disk space savings while minimizing in-memory page space waste. 16K in this example

How to Implement Index Compression

- Pre-measure activities (collect current NLEAF, NLEVELS, HALRBA, HURBA, CPU Utilization Metrics)
- Method 1: DROP INDEX, CREATE INDEX DEFER YES, then REBUILD, and REBIND as needed
- Method 2:
  - ALTER INDEX for correct Bufferpool – this is a deferred change due to size change, not immediate. You will be placed in AREO Pending (SQLCODE +610).
  - REORG INDEX or TABLESPACE (this will change DB2 Catalog bufferpool). Can be SHRLEVEL REFERENCE or CHANGE.
  - ALTER INDEX COMPRESS YES – this change will once again put you in REBUILD pending (SQLCODE +610).
  - REBUILD INDEX. Can be SHRLEVEL REFERENCE or CHANGE.
  - REBIND invalidated packages
  - Unfortunately it won't let you combine the two ALTERs into one.
- Post-measure activities. Evaluate cost/benefits. If negative, undo change.

Index Compression – COMPRESS option

- Be Selective! Only apply INDEX Compression where the benefit will be maximized. That means large indexes (at least top 10% of all indexes in size), not predominantly accessed in a synchronous mode, with very significant DASD savings (50% or higher) predicted by DSN1COMP, and no obvious downsides or risks. Data Warehouse type applications are more likely to benefit than OLTP systems.
- Target based on size and usage and your application knowledge. Pick only the "most suitable" candidates for compression.
- In terms of physical attributes, good candidates have: high values for NLEVELS and NLEAF, a long index key size (more than 9 bytes), often are hierarchically organized multi-column keys, and have a mixture of access types (seq, random, batch, online).
- We don't want indexes which have a 90% synchronous read rate to be compressed. We want to pick indexes from a more blended environment with a mix of synchronous and asynchronous processing. Avoid the small group of indexes where there is little or no pre-fetch activity.
- Pick the buffer pool carefully based on DSN1COMP output: evaluate all your options from the 8K or 16K bufferpools. Avoid 32K.
- BEST PRACTICE: Use COMPRESS YES for all large indexes with large index keys going forward. Prioritorize conversion of the largest indexes first, being sure to keep the DB2 Systems people in the loop for buffer pool planning and advice.

Best Practices Re: CLUSTERing Indexes

- It is always best to define one index per base table as the CLUSTER index. This is because if you don't, DB2 will pick one for you, defaulting to the index with the lowest physical ISOBID value (internal DB2 catalog value). This might not be the index you want or expect. Don't let it default.

- Be sure to pick the index with the most critical, high-volume, set processing. This is often not the PK index, nor the partitioning index. Prior to V8, the partitioning index *HAD* to be a clustering index. In the "old" days, the data had to be clustered by the partitioning columns. Clustering and partitioning are now completely independent decisions for the DBA.

- Sort any batch input transaction files going against the table in the CLUSTER index sequence, whenever possible. If you process unsorted transaction you'll set off a flurry of index splits once your free space runs out.

- Did you know that CLUSTERED indexes are eligible for enhanced Index Lookaside logic in DB2 V9? So are any other indexes with a CLUSTERRATIO > 80% whether defined as CLUSTER or not.

- Consider the APPEND YES option, in light of this, if INSERTs won't naturally be in ascending CLUSTER index order. You then rely on a periodic REORG to restore 100% CLUSTERRATIO.

Comments about Partitioning and Indexes

- The decision to partition and the different types of partitioning is a big subject which is beyond the scope of this presentation.

- PARTITIONING INDEX: Suffice it to say, we strongly discourage the old partition by index approach where the partitioning limit keys were specified in the PARTS VALUE clause of the CREATE INDEX statement, which was the original practice, in favor of table-controlled partitioning. This is sometimes referred to as “Classic Partitioned”. Restriction: You cannot create a partitioning index in a partition-by-growth table space.

- BEST PRACTICE: All new indexes for partitioned tables should use table-controlled, not index-controlled, definition. There are many opportunities for savings to be gained by converting from index-controlled to table-controlled, especially when the partitioning key is limited or not natural, such as Month Number (1-12). We have seen 5-10% reductions in run times of some jobs just by eliminating such trivial indexes.

Comments about Partitioning and Indexes

- PARTITIONED INDEX: Another very important decision when dealing with indexes for partitioned data tables is should the index be similarly partitioned or not? When you specify the CREATE INDEX if you include the PARTITIONED BY Clause you'll end up with multiple index sets, one per partition; otherwise you'll have a Non-Partitioned Index (NPI), where all the index entries for all the data partitions are stored in one index dataset. The vast majority of indexes are NPIs.

- One downside of NPIs is potential performance bottlenecks during parallel insert, update, and delete operations. Another is that an NPI will often negate any advantage of parallelism with utilities at the partition level.

- When you use the PARTITIONED keyword in the CREATE INDEX statement and specify an index key that does not match the partitioning key columns, you'll CREATE a DPSI (Data Partitioned Secondary Index). One index dataset mapping 1:1 to each data partition. DPSI's could not be UNIQUE until DB2 V9. A DPSI is basically a partitioned NPI. There is one index tree structure for each data partition. So a query could potentially have to probe each of these trees.

An Example of Creating a PBR: TS & TB

```
CREATE TABLESPACE TSXYZ IN DBAPP1
  USING STOGROUP SGEXT
  PRIQTY  72000 SECQTY -1
  FREEPAGE 11 PCTFREE 15
  NUMPARTS 12  SEGSIZE 64
  MEMBER CLUSTER
  LOCKSIZE PAGE LOCKMAX 0;

CREATE TABLE APP1.TBXYZ
   (PNUM   SMALLINT,
    COL1   DECIMAL(9,0),
    COL2   VARCHAR(36),
    COL3   DATE)
 PARTITION BY (PNUM)
   (PARTITION 1 ENDING AT (1),
    PARTITION 2 ENDING AT (2),
       …
    PARTITION 12 ENDING AT (12))
WITH RESTRICT ON DROP
 APPEND YES
 IN DBAPP1.TSXYZ;
```

Note: the combination of NUMPARTS & SEGSIZE makes it UTS - PBR

An Example of Creating a PBR: Index

```
CREATE UNIQUE INDEX X1XYZ
  ON APP1.TBXYZ
  (COL1 ASC
  ,COL2 ASC)
  USING STOGROUP SGEXT
   PRIQTY 144000  SECQTY -1
   FREEPAGE 12 PCTFREE 15
   BUFFERPOOL BP16K3
   PIECESIZE 4G
   COMPRESS YES
   NOT PADDED;
```

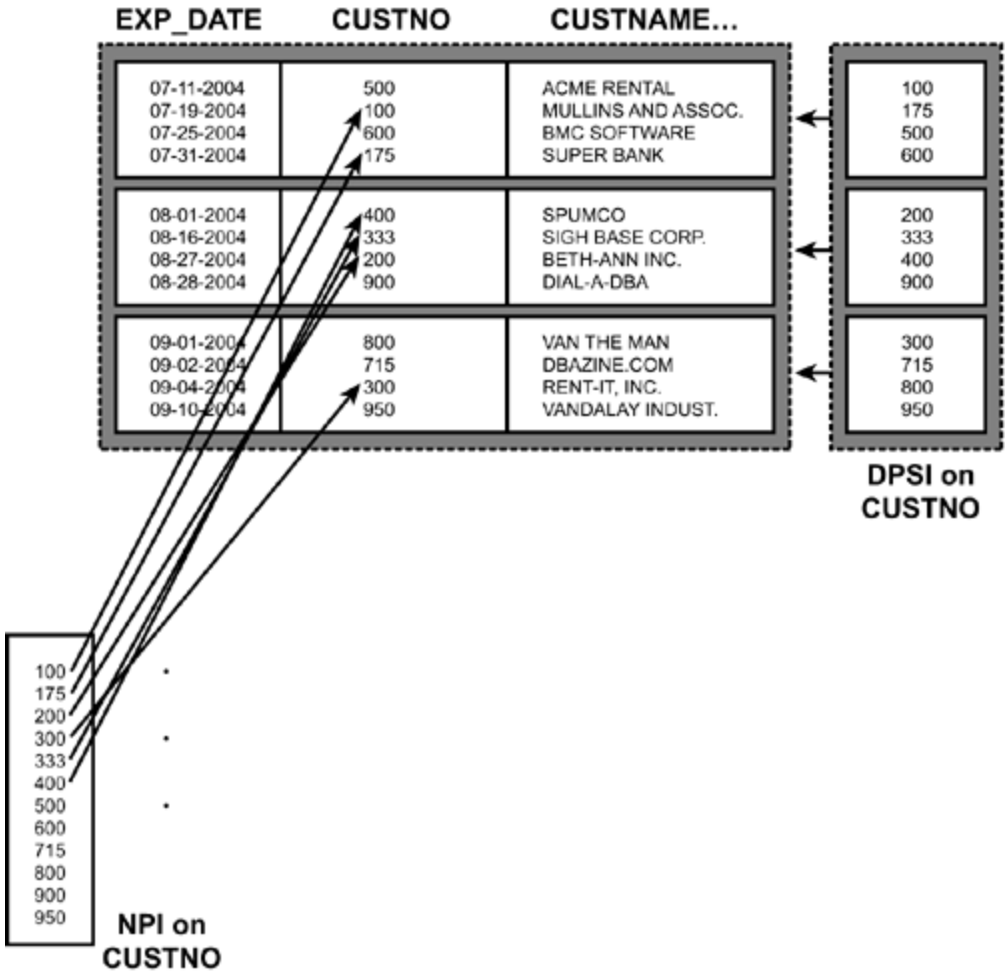
Note: This is the normal case – an NPI – note this is *NOT* a partitioning index

```
CREATE INDEX X2XYZ
  ON APP1.TBRERXYZ
  (COL3 ASC)
  USING STOGROUP SGEXT
   PRIQTY 14400  SECQTY -1
   FREEPAGE 12 PCTFREE 15
   PARTITIONED
   BUFFERPOOL BP3;
```

Note: This is a DPSI because of the PARTITIONED clause and the key definition being different from the table partition key

Data Partitioning Secondary Index (DPSI)

Data table partitioned by 12 months of EXP_DATE



Comments about Partitioning and Indexes

- BEST PRACTICE: The use of DPSI should be definitely strategic and situational. The alternative of an NPI is often the best choice. The use of DPSIs promotes partition independence and therefore can provide a significant performance advantage, especially when the critical performance queries, reference not only the DPSI key columns, but also the partitioning key columns, as well.

- The downside is when the query predicates don't include the partitioning key(s) thus requiring all partition indexes to be accessed. DPSI can be helpful in a DB2 utility strategy emphasizing parallel LOAD utility jobs with the PART option that target different partitions of a table space. For example, utilities such as COPY, REBUILD INDEX, and RECOVER INDEX can operate on physical partitions rather than logical partitions because the keys for a data partition reside in a single DPSI partition. This method can provide greater availability.

- GENERAL RECOMMENDATION: When creating any new partitioned structures adopt either of the two Universal tablespace alternatives PBG (Partition by Growth) or PBR (Partition by Range). You might prefer PBR when a natural or well understood partitioning key is present and PBG when not. UTS is the combination of segmented and partitioned.

- Many new cool DB2 features are available *ONLY* for UTS, such as Clone tables (DB2 9), Hash-organized tables (DB2 10), "Currently committed" locking behavior (DB2 10 -- a means of reducing lock contention); Pending DDL (DB2 10); LOB inlining (DB2 10), and XML multi-versioning (DB2 10 -- required for a number of XML-related enhancements), ALTER TABLE with DROP COLUMN (DB2 11) and this list keeps growing. Also UTS tables have better space maps and space management, INSERT/UPDATE/DELETE logic, partition scope operations (ADD, ROTATE), and are generally more optimized.

Final Thoughts about NPI vs. DPSIs

- DPSIs can also facilitate partition-level operations such as adding a partition or rotating a partition to be the last partition, for those using that strategy.

- You cannot define a DPSI index for a PBG tablespace.

- BEST PRACTICE: If using DPSIs, maintain a very high rate of page range screening aka partition elimination. This happens when the query has predicates on the leading columns of the partitioning key, so that DB2 does not need to examine all partitions. Page range screening can be determined at bind time for a predicate in which a column is compared to a constant; however, it is more often a run time decision, because the column needs to be compared to the value of a host variable, or parameter marker, or special register.

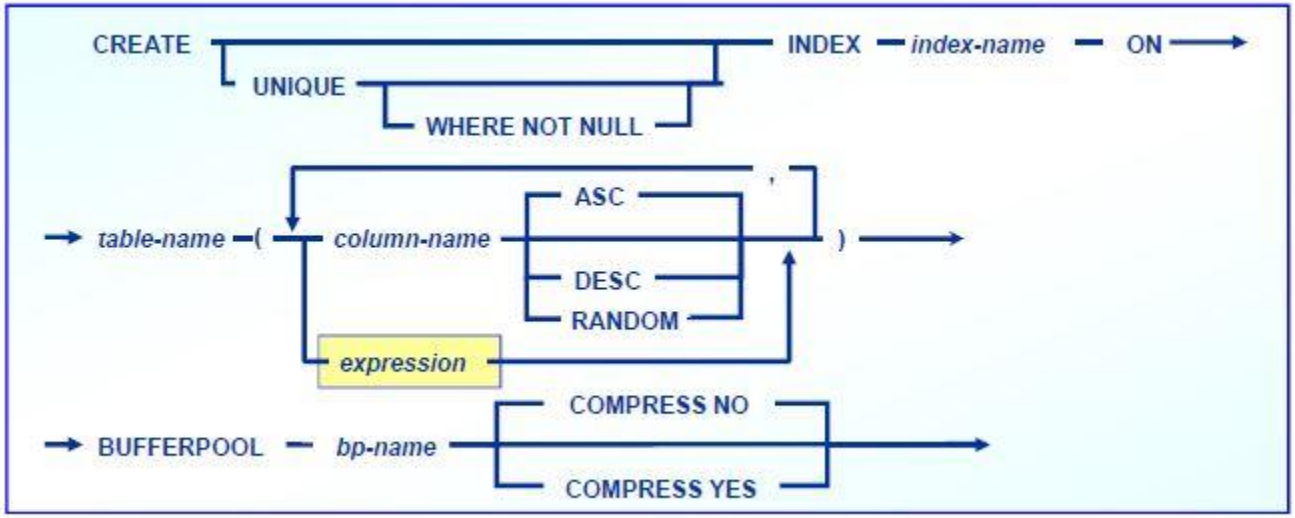
Comments on UNIQUE Indexes

- The decision whether an index should be UNIQUE or not UNIQUE would seem to be application driven. Of course, UNIQUE indexes are more physically efficient, since we save bytes in each index entry since there is only one RID per index record (fixed vs. variable length index record).

- We have found indexes in production defined as not UNIQUE which in fact were UNIQUE. This is a definitional error probably caused by indecision. The cost is probably minor, but there is waste involved.

- Another reason for these might be the case of NULL values invalidating uniqueness. Note: a CREATE UNIQUE INDEX would only allow one row to be NULL, but a CREATE UNIQUE WHERE NOT NULL INDEX will allow any number of values to be NULL, as long as all the NOT NULL values are UNIQUE. This is especially useful when you have a composite index key which contains at least one nullable column, but all non-null entries must be unique.

CREATE INDEX



Index on Expression

- Prior to DB2 V9, we could define index keys only as one or more column name(s) from our table. Then DB2 V9 introduced the Index on Expression option. Basically, you can create an index which is just a simple expression, or a mixture of columns and expressions. The expression can include most functions. But test it out since some things are disallowed.
- An Index on expression can produce orders of magnitude savings when you have a matching expression predicate. Examples:

```
CREATE INDEX a ON tabx (SALARY + BONUS);
CREATE INDEX b ON taby (DEPTNO, UPPER(DEPTNAME));
CREATE INDEX c ON tabz (RTRIM(LASTNAME));
```

- Note that DB2 stores the results of the expressions in the index. The columns themselves used in the expressions are not stored.
- SQL containing the same expressions can utilize these indexes for either data retrieval or index only operations.
- There are some restrictions for indexes with expressions

  Indexes on expression cannot be used for clustering

  DESC/RANDOM may not be specified

  You may not ALTER INDEX add columns to indexes with expressions

  Primary/Foreign keys with expressions are not supported

Index Hot Spots and the Solution

- A “Hot Spot” is a page which has a very high update rate. When applied to an index page, especially a non-leaf page it can mean contention and reduced parallelism translating to poor response times for users.
- Imagine the situation caused in the index, while inserting ascending high keys, perhaps because a timestamp is on the leading edge of the key – this means all inserts will go to the end of the index. In Data Sharing environments, this causes p-lock contention to occur, and multiple transactions may contend for the index page and some will have to wait to get the page.
- Asymmetric index splits help reduce the splits when inserts come in order (ascending or descending), but it can still be a bottleneck when multiple access is needed. In DB2 V9, they introduced the RANDOM option to the CREATE INDEX statement as an alternative to ASC or DESC as a column modifier. So if you create an index on LAST_UPDATE RANDOM, internally DB2 will scramble the values in the LAST_UPDATE column so it will be inserted in a quasi-random manner not all at the end. The scrambling is reversible of course – so we can test for equality – index only access is still possible.
- The drawback here is you’ve lost the ability to do a sequential scan on that index. For example, consider an ORDER BY LAST_UPDATE. Normally we would hope to avoid a sort by using the index but DB2 can’t do that since the physical rows are not in logical ascending order - a sort of the data rows will be necessary.
- For some reason, RANDOM and NOT PADDED are incompatible choices.
- BEST PRACTICE: Only use RANDOM DB2 indexes where INSERTS bunch up causing noticeable hot spots, and where sequential scans on this index are not necessary. Test carefully as performance can accelerate or degrade dramatically.

Index PCTFREE and FREEPAGE parameters

- DB2 indexes default to PCTFREE 10 and FREEPAGE 0. This is too much for some indexes, and too little for others. The price you pay for reserving insufficient expansion space in your index is overhead in the form of Index Splits, which can be quite expensive, especially in a Data Sharing environment. Index splits can involve one or more non-leaf index records as well, sometimes holding a lock or latch on a high level index record impairing concurrency.
- DB2 V9 improved matters with Asymmetric index Split Logic. In general, view free space of both types as needed to efficiently handle inserts, and some updates. I would say in general you should anticipate how much percentagewise will be inserted or updated between expected REORG periods (weekly, monthly, etc.) and code accordingly.
- PCTFREE can be 0 thru 99. The value PCTFREE 15 means leave 15% of every index leaf page free. Load/Rebuild/Reorg entries up to the 85% high water mark and then skip to the next index leaf page. PCTFREE only refers to leaf pages – non-leaf pages always have a 10% PCTFREE. Regardless of the value of PCTFREE you are always guaranteed that one index entry will fit per leaf page.
- FREEPAGE can be 0 thru 255. The value FREEPAGE 15 means one free page is left for every 15 regular index pages, so 14 will contain index entries, leave one index page completely empty to accommodate a future index split. You should leave a few of these, at least one every 64 pages. When DB2 has to split an index page, it will use one of these “spare” index pages as long as it can find it within 64 pages of the split location. Then it will be counted as “LEAFNEAR” not “LEAFFAR”.
- You cannot specify PCTFREE nor FREEPAGE, however, for an index on a DECLARED GLOBAL TEMP TABLE.
- Of course if your table is SELECT only, specify PCTFREE 0 and FREEPAGE 0.
- BEST PRACTICE: For updateable tables subject to growth, specify sufficient PCTFREE and FREEPAGE in indexes as well as data, so that REORGs can be infrequent, perhaps monthly. Avoid excessive number of index splits. Proactively monitor SYSIBM.SYSINDEXPART- LEAFDIST,LEAFNEAR, LEAFFAR, NEAROFFPOSF, FAROFFPOSF.

Asymmetric Index Splits

- In older versions of DB2, once an index LEAF page became full, DB2 would grab an available empty index page, and divide the entries 50-50% thus "splitting" the index page down the middle, moving half of the entries to the new page. This was known as a "symmetric split". Of course in many situations, such as a key containing a date or timestamp, this is problematical as at least one of the two pages might remain 50% empty forever.

- Then DB2 got a little smarter and introduced the 100/0 index split, or Asymmetric Splits). In this scenario no existing entries are moved to the new page. This works best for LOAD and ascending key INSERTs.

- This logic was improved again in DB2 V9. DB2 automatically detects ordered index insertions and performs asymmetric index splits. A clever algorithm remembers an "insert-range" so it will treat quasi-ascending keys correctly. Pages are left more than half full (but not necessarily 100% full). The number of index page splits was arguably reduced by 50% in most studies.

Index PIECESIZE parameter

- DSSIZE is to tables what PIECESIZE is to indexes. It is a theoretical maximum size which one physical index dataset can reach. PIECESIZE can be any multiple of K, M, or G. Larger values for PIECESIZE > 2GB will require extended DASD Storage Groups. PIECESIZE is in addition to, and has no effect on primary and secondary space allocation for the index. PIECESIZE is a specification of the maximum amount of data that an index data set can hold, and not the actual allocation of storage itself.

- Be careful not to code a small PIECESIZE and a large primary space allocation for the index. That excess will be wasted. Ideally the value of your primary quantity and secondary quantities should be evenly divisible into PIECESIZE to avoid wasting space.

- PIECESIZE will control how many datasets are created, and what the maximum size index you can reach will be.

- For example, most of us in the past have created NPI indexes and never bothered to specify PIECESIZE – what happened all this time? Let's just assume the underlying base tablespace also omitted DSSIZE and that the page sizes are 4K. The default PIECESIZE would be 2GB and your maximum NPI index would be 64GB (32 datasets times 2GB). Remember that 32 datasets is the limit if the underlying tablespace isn't defined as LARGE or has a DSSIZE > 2GB.

- Default formula PIECESIZE = MIN (x, 4G / (MIN(4096, 4G /(x/y)))*z)

  Where x is the DSSIZE of the associated table space
  Where y is the page size of the table space
  Where z is the page size of the index
  and Where 4G is 2 to the 32nd power

- But if you specified PIECESIZE as 32GB you could have a maximum of 4096 datasets x 32GB for a total of 128 TB.

Index BUFFERPOOL parameter

- Specifying BUFFERPOOL indirectly indicates the index page size. Prior to DB2 V9, you had to choose one of the available 4K bufferpools, but now you can also choose BP8K, BP8K1-9, BP16K 0-9, or BP32K0-9. There are several good reasons to do so. Example:

```
CREATE INDEX   XYZ.X1CUSCTL on XYZ.TBCUSCTL
      …..
      BUFFERPOOL  BP8K3
      …..
```

- If BUFFERPOOL parameter is omitted, DB2 looks at the database definition for a default index bufferpool. If omitted there, it looks at an installation panel for a generic default, otherwise it uses BP 0.

- You can ALTER INDEX to change an index's BUFFERPOOL assignment thus changing its page size. This sets that index into REBUILD PENDING. Be aware that the change in bufferpool doesn't take effect until the next time the indexspace is stopped and re-started.

```
ALTER INDEX XYZ.X2EMPTBL
      BUFFERPOOL BP16K3
```

Potential Benefits of Increasing Index Page Size

- Bigger index pages means more index entries per page hence better fan-out
- The number of levels will often be reduced, especially for largest indexes, thus fewer getpages.
- Larger index page sizes can also reduce index splitting activity (especially good for "ordered" key inserts). Index leaf page splits are especially painful for indexes with GBP dependencies (data sharing).
- Non-Leaf bufferpool space consumption is reduced. Consider this example (16 byte key):

Index Page	Entries per Leaf	Number Leafs	Levels	Fan-out	Non-Leafs in BP	Bufferpool Savings
4K	168	5,952,381	5	158	148 MB	--
8K	338	2,958,580	4	318	72.9 MB	51%
16K	680	1,470,589	4	639	36.1 MB	76%
32K	1362	734,215	3	1280	18 MB	88%

- Benefits can be even greater with larger index keys.

Larger Index Page Size Downsides

- Of course there are tradeoffs. Consider random selects of one row. They will take slightly longer due to the larger leaf page. The I/O operation takes longer, and more bufferpool space is used to contain leaf pages.
- In high update environments, you may hit more contention on some leaf pages because more entries are packed inside – thus "hotter" pages, which can be good and bad.
- So as in most cases, it all depends …
- Positive indicators for larger index page size: workloads with many scans and multi row access. Large indexes with many levels, applications suffering from index page splits.
- As in all cases, approach any change cautiously and do adequate testing in lower lanes before moving to production.

Miscellaneous Index Best Practices

- Keep the number of Indexes as Low as Possible. Loading IM rows with 1 index takes about 25 seconds. Loading the same size table with 5 indexes takes about 50 seconds.
- Consider having no indexes at all, or not more than one UNIQUE Primary Key index, and possibly one JOIN enabler index for small tables (less than 100 pages).
- Remove Indexes Not Used as per SYSINDEXSPACESTATS.LASTUSED (V9). This includes static and dynamic usage.
- Create indexes with DEFER YES. Use LOAD or REBUILD INDEX to populate later for a more efficient process.

Thanks

Q & A Anyone?

Please contact the author of this presentation, Hal Steiner, with any questions, concerns, comments, or corrections.

harold.steiner_iii@bankofamerica.com

732-673-6522
@HalSteiner3

.