Dan Luksetich is a Db2 DBA consultant. He works as a DBA, application architect, presenter, author, and teacher. Dan has been in the information technology business for over 35 years, and has worked with Db2 for over 30 years. He has been an application programmer, Db2 system programmer, Db2 DBA, and Db2 application architect. His experience includes major implementations on z/OS, AIX, i Series, Windows, and Linux environments. His industry experience includes retail, banking, fraud detection, analytics, government, and utility. He specializes in highly available, high-volume transaction processing against very large database systems.

Dan's experience includes, but is not limited to:

- Application design and architecture
- Business analytics
- SQL consulting and education
- SQL coding and application programming
- Database administration
- Complex SQL
- SQL tuning
- Db2 performance audits
- Replication
- Disaster recovery
- Stored procedures, user-defined functions, and triggers
- Db2 REST services

Dan likes beer and is a Certified Cicerone!

What is an Anti-Consultant?

DanL Database Consulting

- Db2 for z/OS System Programmer, Database Administrator, Application Architect, Application Developer, Author, Teacher, Lecturer, Advocate!
- Db2 for LUW Database Administrator, System Administrator, Application Developer.
- IBM Gold Consultant, IBM Champion for Analytics, former IDUG Content Committee Chairman, IDUG DB2-L Administrator, IBM certification test developer.
- Inventor of the first Db2 podcast series "The Db2 Cocktail Hour" beginning in March of 2005 with Susan Lawson.
- Co-inventor of IDUG "Fun with SQL" with Kurt Struyf.
- Created the "world's most complicated SQL challenge" only ever solved by 2 people (neither were me)!
- Still available, still free, although slightly outdated: The Db2 for z/OS Performance Handbook. https://www.ca.com/content/dam/ca/us/files/technical-document/the-db2-for-zos-performance-handbook.pdf
- What do I do now? I design and build really large high-volume database transaction processing systems.
- I am a Level 2 Certified Cicerone®.
- Who is Coatcheck? He is an internationally recognized "open mic night" singer and comedian having performed in many different cities and countries.

When I has a child, I wanted to be a meteorologist. When I was a teenager, I wanted to be a rock star. When I was in college, I wanted to be a computer programmer.

Now, I want to be a brewer…or a short order cook…or maybe a BBQ pitmaster!

The word Cicerone (sis-uh-rohn) designates a trained professional, working in the hospitality and alcoholic beverage industry, who specializes in the service and knowledge of beer. The knowledge required for certification includes an understanding of styles, brewing, ingredients, history of beer and brewing, glassware, beer service, draught systems, beer tasting, and food pairings. To claim the title of Cicerone, one must earn the trademarked title of Certified Cicerone® or hold a higher certification. Those with a basic level of expertise gain recognition by earning the first-level title Certified Beer Server. Only those who have passed the requisite test of knowledge and tasting skill can call themselves a Cicerone.

And just for the record an anti-consultant is someone that primarily wants to solve problems, with money only being an added bonus to the effort. An anti-consultant's main objective is to solve your problems, and give you the knowledge such that you never need his or her service again. To summarize, an anti-consultant is always trying to put themselves out of work. I've failed in that aspect for the last 25 years.

Disclaimer

- I am NOT a Db2 expert!
  - There are NO experts as the breadth of features, changes in software and hardware, are too vast for one person to keep pace
  - You should consider the advice in this presentation as just that, advice!
  - Your results can and will vary
  - Only empirical evidence will be the determining factor regarding performance in each situation
  - TEST, TEST, TEST
    - This is much easier than imagined
- Read the notes pages as there is more information than what is in the slide (and what I will surely forget to mention during the presentation)
  - We will make this available on the Db2Night site or you can email me

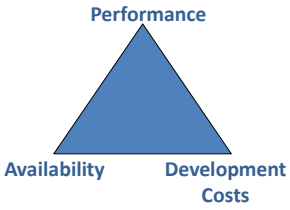
I worked for 10 years for an electric company in Chicago. For 5 years I was an Assembler and COBOL programmer, and for another 5 years I was a Db2 System Administration and Database Administrator with a little IMS thrown in. I did a significant amount of tuning, but could not convince management to do more of it, until…

"Big Name Consulting" was brought in to "modernize" our operations. They assigned a "Db2 expert consultant" to work with me on my long requested tuning project. I was introduced and the first thing the kid said to me was "what is Db2?" I worked with him teaching him as I tuned a database and application, and after success he went away. Then my boss told me that "Big Name Consulting" was going to bring in another "Db2 expert consultant" to work with me on a second tuning project…

I went to my desk and typed a letter of resignation. Thus began my career as an anti-consultant.

Performance Overview

- Do you want high performance? Low operational costs?
  - It must be part of your design
  - Decisions need to be made, and management needs to support performance
  - Performance is NOT an afterthought
- Performance choices during design
  - Physical object design (tables, table space, indexes)
  - Application design with consideration of physical object design
    - High read, high update, or high insert?
- Testing the performance choices is critical



There are a myriad of tuning possibilities when it comes to the database. Before jumping onto any of the touted performance features of Db2 it is critical to first understand the problem that needs to be solved. If you want to partition a table, are you actually hoping that the partitioning results in a performance improvement or is it for data management, availability or flexibility? Clustering of data is critical, but clustering has to reflect potential sequential access for single table, and especially multi-table access. Generic table keys are OK, but clustering is meaningless unless parent keys are used for clustering child table data when processing follows a key-based sequential pattern or when multiple tables are being joined. Natural keys may be preferred, unless compound keys become exceedingly large. There are a number of choices of page sizes, and this can be especially important for indexes where page splitting may be a concern. Tables can also be designed specifically for high volume inserts with such features as “append only” and “member cluster”.

There is no reason to sit in meeting rooms for hours or days trying to come up with a database design that you “think” will perform properly. I personally let the database itself influence the design. This involves setting up test database designs and test conditions that will mock the predicted application and database design. This usually involves generated data and statements that are run through SPUFI, REXX programs, db2batch, or shell scripts, all while performance information is gathered from monitors and compared. Sound complicated? It really isn’t, and most proof-of-concept tests I’ve been involved with have not lasted more than a week or two. The effort is minimal, but the rewards can be dramatic.

Index Design Choices - Overview

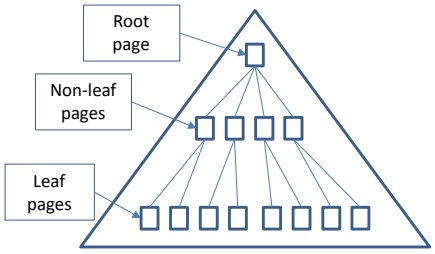
- There are multiple basic index design choices for tables
  - Partitioned index
    - Including partitioning index (deprecated)
  - Nonpartitioning index
    - Data-partitioned secondary index
    - Nonpartitioned secondary index
- Other indexing options
  - Index on expression
    - Including JSON indexes
  - XML indexes
- These index choices need to be explored in detail when making design decisions
  - This presentation only touches on this as we explore design issues relative to performance choices within indexes

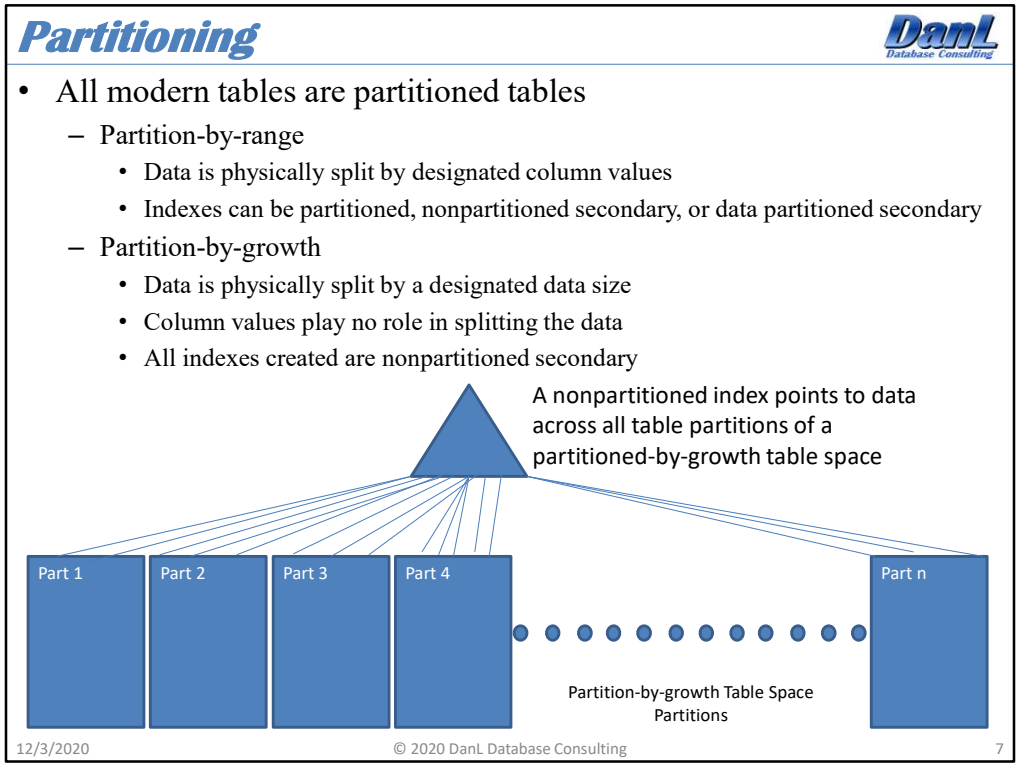
Although this presentation is not necessarily about choices concerning the type of index I should note that it is an important overall design decision. After all, there is only so much you can do within a 60 minute presentation. That being said I do want to point out one very specific design choice: partitioned or not. This, of course, assumes that the underlying table space is partitioned. Actually, all table spaces are partitioned with the choices being partition-by-growth or partition-by-range. If the choice for table space is partition-by-growth then the only type of index that can be built is a nonpartitioned secondary index. The choices become more complex when the underlying table space is a partition-by-range table space.

A partitioned index is one where the leading columns of the index key matches the partitioning columns of a partition-by-range table space. A nonpartitioning index is one where the leading columns of the index keys do not match the partitioning columns of a partition-by-range table space.

Db2 indexes

- Db2 indexes use an optimized B-tree implementation
- A b-tree index stands for “balanced tree” and is a type of index that can be created in relational databases
  - 1 to n levels depending upon the number of index entries
  - Non-leaf pages (including root) have pointers depending upon key values that point to other non-leaf pages.
  - Leaf pages contain keys values and the associated RIDs (row identifiers) that point to rows on data in the corresponding table

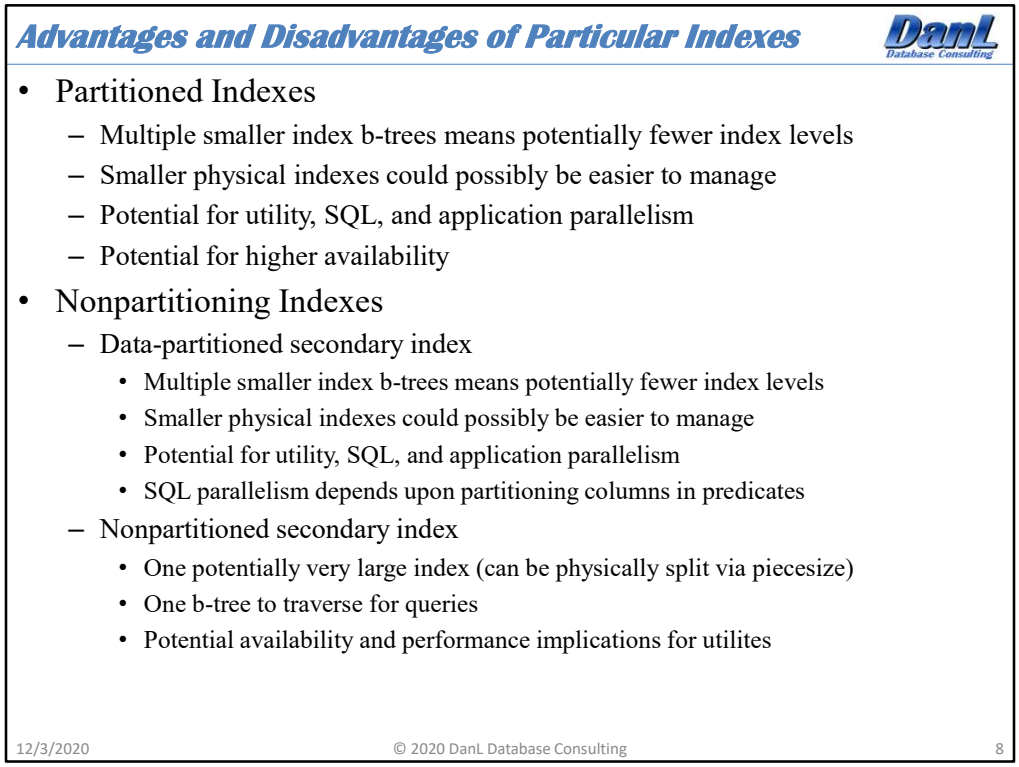




With a partition-by-growth table space only nonpartitioned secondary indexes can be created. So, a very important table space design decision is to partition by range or not. There are several reason to create a partition-by-range table space:

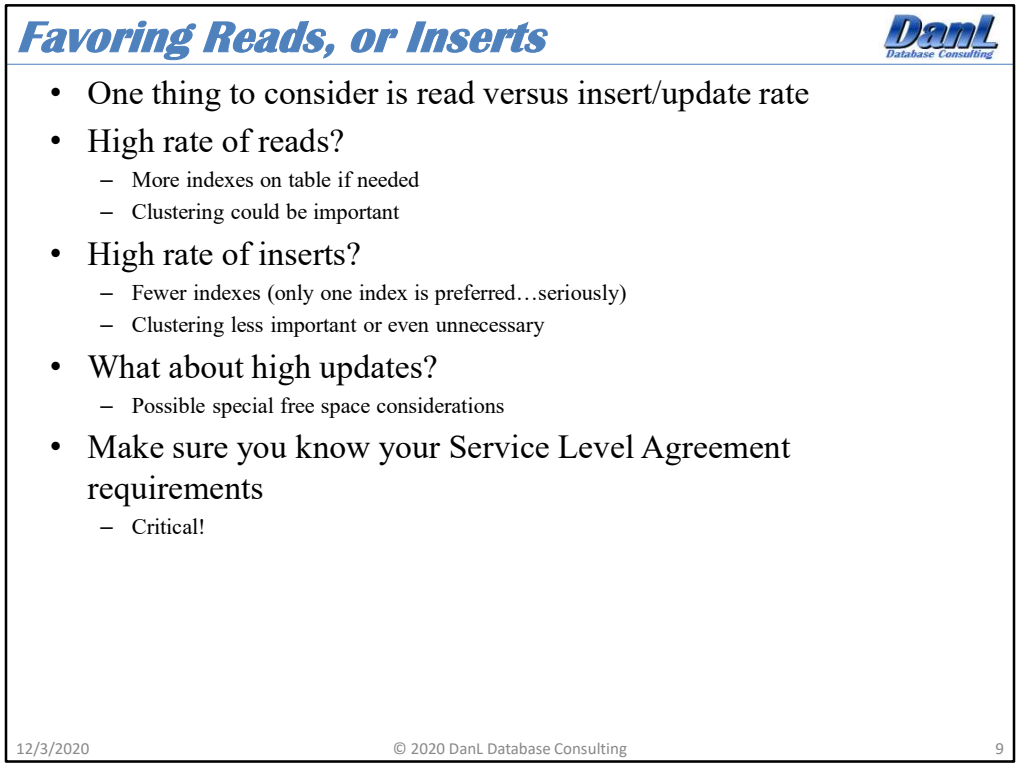
- Availability – perform maintenance on a specific partition while other partitions remain available
- Performance – potential for enhanced parallelism for SQL, utilities, and applications
- Manageability – it may be simpler or more efficient to support several small objects versus one very large object

My personal preference is if I have specific availability or performance requirements, or a table is going to be extremely large then I am likely to choose a partition-by-range table space.



This is a complex topic and very specific to particular application and database design considerations. You can simplify by asking yourself some specific questions, such as how many objects do you want to manage? Do you want to manage very large objects? Can you develop a partitioning scheme? What are the application requirements?

One note about data-partitioned secondary indexes



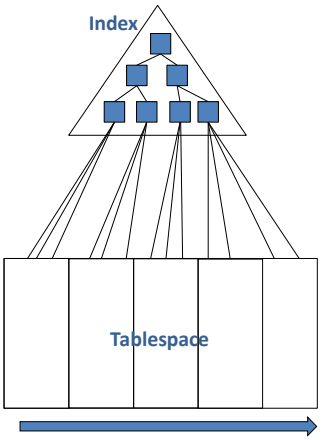
When designing a table for a high performance application there are many things to consider. However, a simple question should be answered when designing a table or set of tables; Is this a high insert table or a high read table? To put it another way are you designing for fast insert or optimal read performance? Once this is answered then the design path can be chosen for the table in question. You must get confirmation from the development team and management. If the boss says she wants the fast insert option along with the fast read option make sure she understands that it is only a dream. She must favor one or the other. Get a service level agreement for performance.

I once asked a manager how fast he wanted the application to run. His response "I want it to be as fast as possible". My response was "Oh great that's easy, buy an infinite number of machines!" After the initial shock he then gave me a real number!

e.g. 90% of all transactions with a response time under 500ms.

Clustering

- Db2 attempts to keep table rows in order of the clustering index
  - Index keys are always in order
  - Important for sequential readers
  - Db2 can take advantage of performance enhancers
    - Dynamic prefetch
    - Index look aside
  - RELEASE(DEALLOCATE) packages or high performance DBATS




```
SELECT <columns>
FROM   TABLE1
WHERE  <clustering key column> >= ?
ORDER  BY <clustering key column>
```

12/3/2020 10

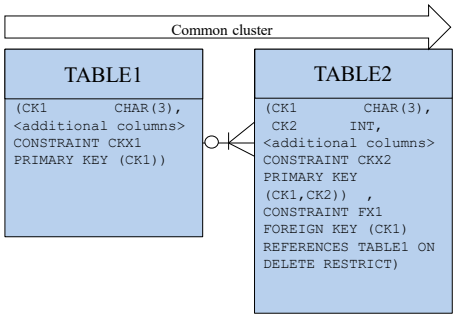
Db2 can achieve incredibly high read rates in support of batch processing when a sequential reader (usually the driving cursor for a process) is reading from a table in the order of the clustering key and the table is reasonably well organized. Depending upon insert and update activity and available free space in the table space, REORGs may be required at certain intervals in order to maintain this high efficiency. The cursor fetching the data should use the clustering key in the predicate, and in the ORDER BY clause, to ensure the best performance.

Clustering

- A common clustering is important for joins and for random readers in general
  - If queries are returning multiple rows
  - Also good for large sequential readers
    - Materialization may be an issue (it always depends)

```
SELECT <columns>
FROM   TABLE1 T1
LEFT OUTER JOIN
       TABLE2 T2
ON     T1.CK1 = T2.CK1
WHERE  T1.CK1 = ?
ORDER  BY T1.CK1, T2.CK2

CK1 = <clustering key column>
```



A common clustering between two related tables can be beneficial for random readers as well as most sequential readers. A good example is a table relationship between a parent table and child table that is identifying. Identifying relationships exist when the primary key of the parent entity is included in the primary key of the child entity. On the other hand, a non-identifying relationship exists when the primary key of the parent entity is included in the child entity but not as part of the child entity's primary key.

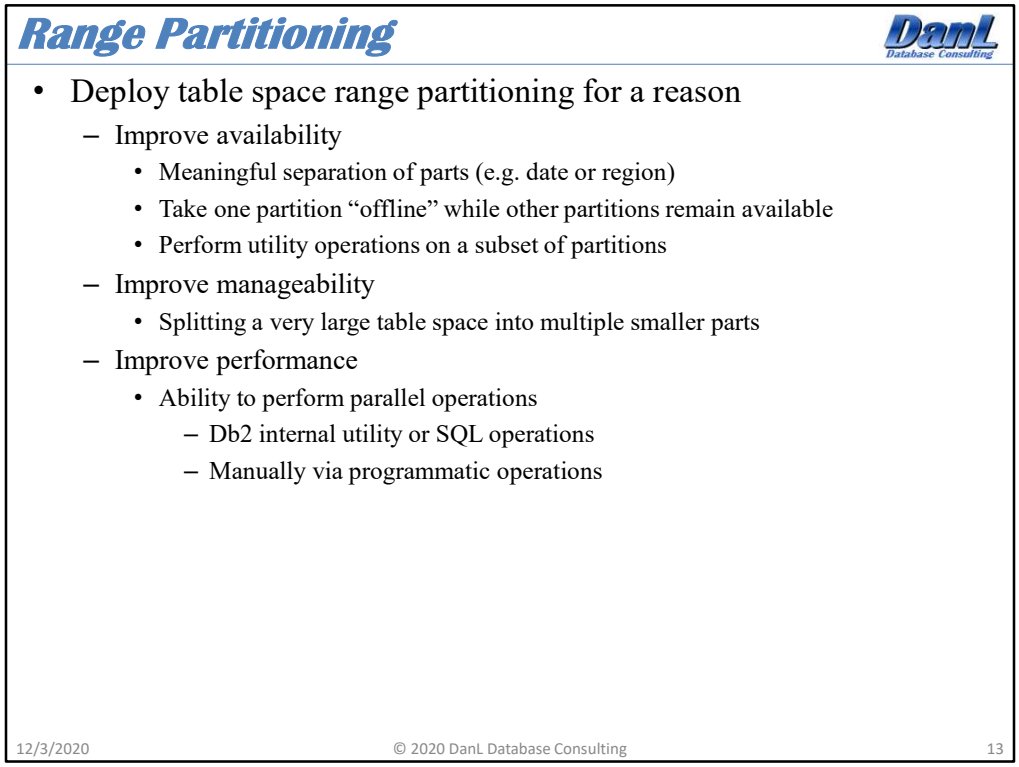
When traversing the index of the inner table (TABLE2 in this example) via the primary key index, key values for multiple rows will be co-located. If TABLE2 is well organized then all of the rows related to TABLE1 will be clustered together. This will minimize the number of random read operations on behalf of the query. Certain things, such as the bind parameter RELEASE(DEALLOCATE), including for high performance DBATS, can also enable the built-in performance enhancers sequential detection and index look aside.

Maintaining Cluster

- Maintaining cluster for a table that receives inserts requires free space
  - PCTFREE setting at the table space or partition level
  - FREEPAGE setting instead of PCTFREE at the table space or partition level ONLY when row length two long for more than 1 row per page
- Set PCTFREE based upon insert rate prediction for uniform distribution
  - Percentage of inserts based upon table partition size (number of rows)
  - Desired number of days between REORGs
  - This is a "starting point" for determining free space
- Set PCTFREE FOR UPDATE if there are variable length rows that increase in size
- PCTFREE + PCTFREE FOR UPDATE = total % free space

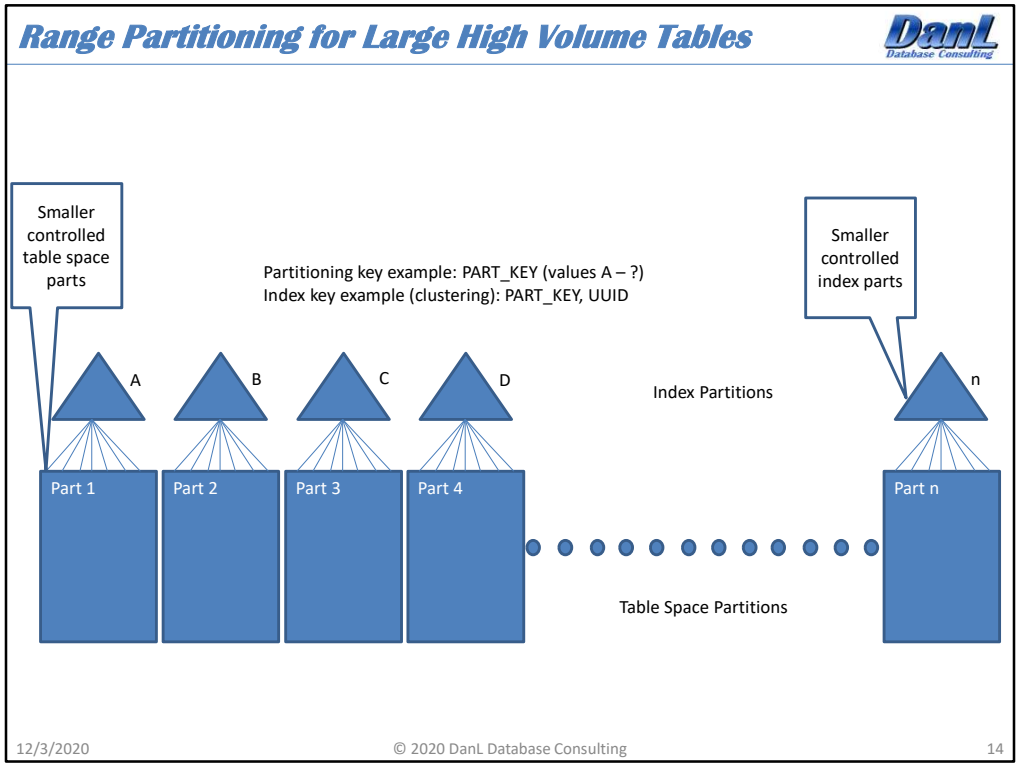
<predicted # rows inserted during n days between REORG> / <# of rows in table or partition> = PCTFREE

If clustered data is desired or required to support sequential processing of table data, and the table receives frequent inserts then proper free space is required to maintain a certain level of cluster. Assuming an even distribution of data in the table space relative to the clustering order then a simple formula can be used to calculate an appropriate PCTFREE setting for the table space and/or table space partition. By using the insert rate and current table size in number of rows a simple formula can be used to determine the PCTFREE setting in order to accommodate the new data in between REORGs. If the distribution of data is skewed then adjustments will have to be made depending upon that skew.



There are two choices for partitioning; partition-by-growth (PBG) or partition-by-range (PBR). Typically a PBG table space is used to store relatively smaller and more stable table data. For really large tables and/or tables with high volume processing the preference is to use PBR table spaces. While a PBG can support partition level utility operations and parallelism, the DBA has little control over the distribution of the data in the partitions. In addition, a partitioned index cannot be created for a PBG table space. For a PBR table space, if a meaningful partitioning key can be defined, then there are definitely advantages to using a PBR table space for large high volume tables. Some advantages include:

- Meaningful partitioning keys
- Query parallelism
- Ability to control application parallelism based upon partitioning key values
- Partitioned indexes

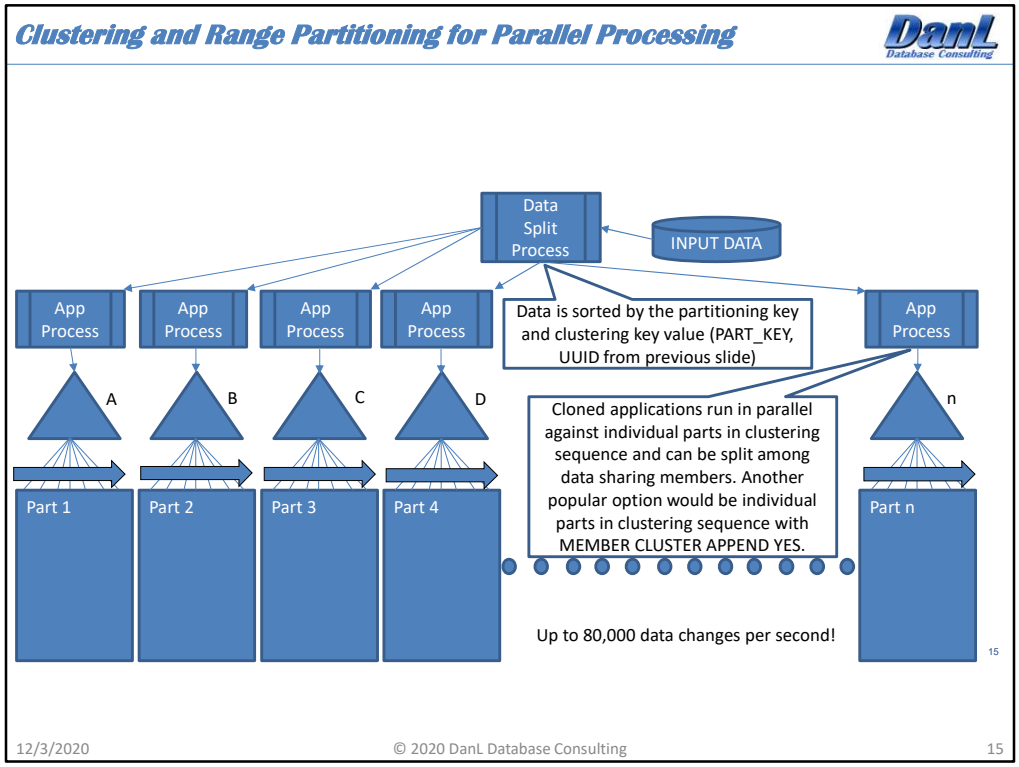


There are several reasons for choosing range-partitioning:

- Rotating and aging data
- Separating old and new data
- Separating data logically for region or business segment based availability
- Spreading data out to distribute access
- Logically organizing data to match processes for high concurrency

One of the performance advantages to range-partitioning is in distributing the data to spread out overall access to the table space. What this means is that index b-tree levels can potentially be reduced due to reduced entry count per partition and that can reduce I/O's and getpages to indexes. In addition, free space searches could be more efficient in a partitioned situation since the data is broken up and the searches are on a partition-by-partition basis. The same is true for any partitioned indexes defined against the table space.

From a performance perspective, range-partitioning is useful to enable a high degree of parallelism for queries and utilities. In addition, queries coded against range-partitioned tables can take advantage of partition elimination, also sometimes called page range screening, when predicates are coded against partitioning columns. These predicates can contain host variable or parameter markers, literal values, and, beginning with DB2 11 for z/OS, joined columns.

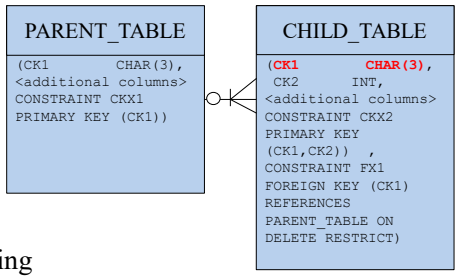


For the highest level of partitioned table access possible an effective design technique to use is called “application controlled parallelism”. This is a technique that is useful when there exists high volume batch processes acting against very large OLTP data stores. In these situations, it is a careful balance between partitioning, clustering (or intentionally not clustering), and distribution of transactions in a meaningful way across multiple application processes. What I mean by a meaningful way is that the distribution of inbound transactions has to match the partitioning and/or clustering of the data in the table space partitions accessed during the processing. In this way, each independent (or sometimes dependent) process operates against its own set of partitions. This will eliminate locking contention and possibly allow for a lock size of partition and use of MEMBER CLUSTER to further improve performance of ultra-high insert/update tables.

When the input is clustered in the same sequence as the data is stored, or you are using MEMBER CLUSTER or APPEND ON to place data at the end of a table space partition, Db2 can take advantage of sequential detection to further improve the performance of these types of processing. In these situations (real world) the business transaction rates have been as high as 21,000 per second, single table insert rates as high as 30,000 per second, and application wide insert rates as high as 60,000 per second with overall data change rates at 79,000 per second. These are not test cases but real world results.

Index Performance Considerations

- First rule of index design is to build only the indexes that are needed
  - For transactions!
  - Read-only? Create as many indexes as needed
- Start with one index!
  - Primary key
  - Clustering
  - Range-partitioned
  - Foreign key support
- What does this mean?
  - Use natural primary keys
  - Define relationships that are identifying
    - Compound primary key of child table
    - First part of Primary key is foreign key
  - Use INCLUDE columns if possible to avoid additional indexes



12/3/2020 16

If a table is read-only then there is no need to worry about indexes as no number of indexes will create a negative performance situation. However, most tables are not read-only. A common rule of thumb might be 80% reads and 20% data changes, however in this day of mass ingest those percentages could be highly inaccurate. It is important to note that indexes do not get updates, and an update to an index is a delete and insert of the index entry. It is also very important to know the data change rate for tables, as well as the common access paths into those tables. Then a decision can be made about the number of indexes on the table. However, a very good rule of thumb is to start with a design that utilizes only one index per table! The goal is to have this index serve all purposes, whether it be clustering, partitioning, access path, etc.

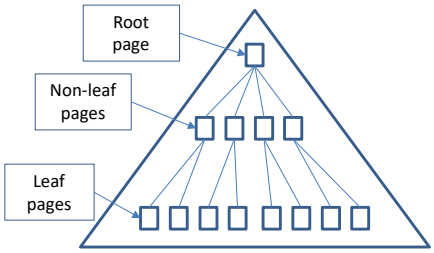
Use identifying relationships when setting up tables. In database terms, relationships between two entities may be classified as being either identifying or non-identifying. Identifying relationships exist when the primary key of the parent entity is included in the primary key of the child entity as the initial columns of that key. On the other hand, a non-identifying relationship exists when the primary key of the parent entity is included in the child entity but not as part of the child entity's primary key. A significant number of modern day developers will object identifying relationships as they prefer a single column system-generated unique key for each table. So, not every table needs to be designed in this manner, but perhaps a compromise can be reached with only the highest volume data changes tables getting the 1-index design.

Create Indexes Only When Necessary

- Every additional index introduces a random reader
  - For inserts
  - For deletes
  - For updates that change the value of an indexed column
    - Indexes aren't updated, only deletes and inserts
- System generated keys contribute to index proliferation
  - GUIDs, UIDs and UUIDs
  - Identity columns and sequence objects
- Identifying relationships help
  - Even for the system generated keys (more on this in a few slides)

Be careful in creating additional indexes on a table, especially for tables with a high insert or delete rate, and also for tables in which the indexes keys would be heavily updated. For partitioned tables, non-partitioned secondary indexes and data-partitioned secondary indexes can provide further challenges in the form of large size and/or access path challenges.

Index PCTFREE and FREEPAGE



Recommendation only for natural key or random index (new keys non-ascending or descending).

- Unlike table spaces, indexes are always kept in key sequence
- PCTFREE should be utilized to avoid index splits between index REORGs
  - Calculate the number of new index entries between index REORGs divided by the total number of index entries to estimate PCTFREE. The goal is no index page splits between index REORGs.
- FREEPAGE should be utilized if index splits can't be avoided between index REORGs

When an index page fills with entries it must be split. When Db2 splits an index page it needs to put the new page somewhere, and it needs a completely empty available page to do it. Another thing that happens during an index page split is that Db2 will serialize on the root page when the split occurs. So, index page splits are a serial process, meaning one agent at a time can split a page. For highly parallelized processes where you have multiple agents acting on a single index or index partition then there could be concurrency issues. Since indexes are typically maintained in key sequence then free space should be allocated such that index page splits are avoided between index REORGs. If splits can't be avoided then a combination of PCTFREE and FREEPAGE can be set to try to reduce the overhead of splits so that if one happens Db2 can find a page nearby to use for the new page. This is most important if the index key is a natural key.

If the index key is a system-generated ascending or descending value, or if the key is non system-generated but still ascending or descending in nature (that is, new keys are assigned in a sequence), then it is perhaps best to allocate no free space and let Db2 simply tack new entries to the end of the index. In this situation if the index keys are updated then some free space might be helpful.

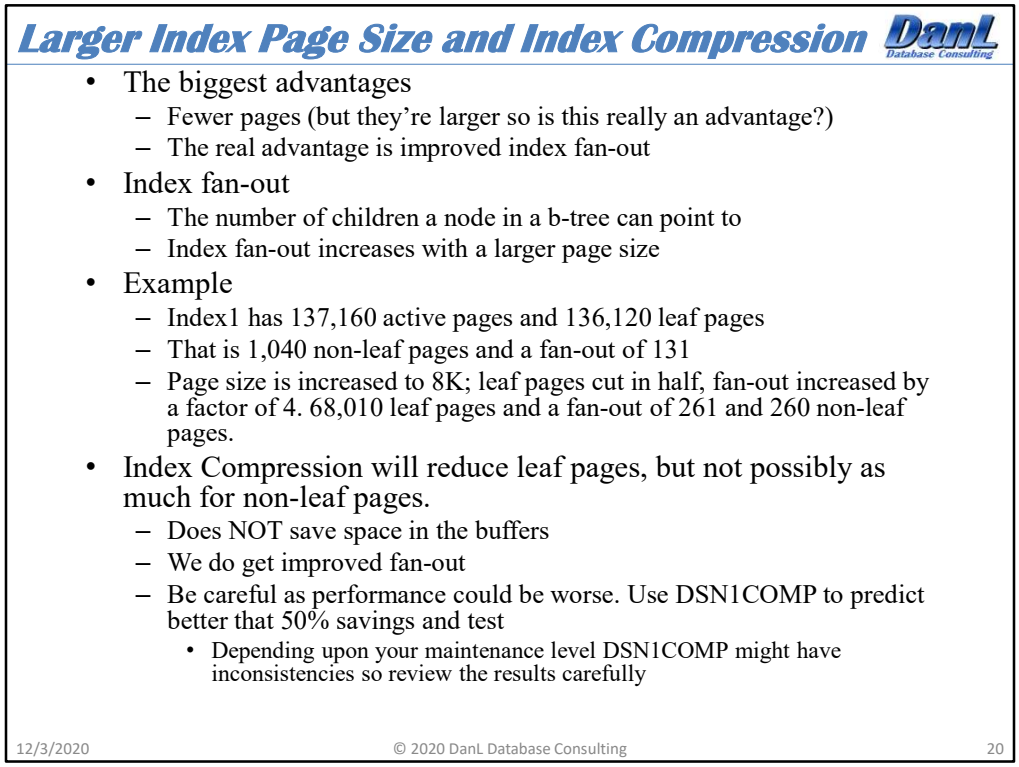
Indexes should be reorganized more frequently than table spaces.

Really Large Random Index Challenge

- A really large 4K page index that has random access can present a performance challenge
  - Leaf pages are very distant from each other
  - Small subset of pages are in the buffers relative to index size
  - Index lookaside is irrelevant
  - DASD performance is impacted by excessive synchronous I/O
  - Excessive index writes can be an I/O response impact
- The solution may be a larger index page size
- A combination of large index page size plus index compression is also worthy of investigation
- One Example: very large random index for system-period history table consumes 12% of I/O and I/O response time for 9% of SQL activity. These are simple inserts…

When an index allocated with 4K pages becomes quite large, at 200+ gigabytes in size, they can become a bit of a performance impact. This is due to the fact that there could be significant random access to the index, and that could lead to a large volume of synchronous I/O, and very inefficient I/O's. These indexes are difficult to tune because as free space is added in the form of PCTFREE and FREEPAGE it increases the size of the index, and that can exacerbate the situation.

This may be the perfect situation for a larger index page size!

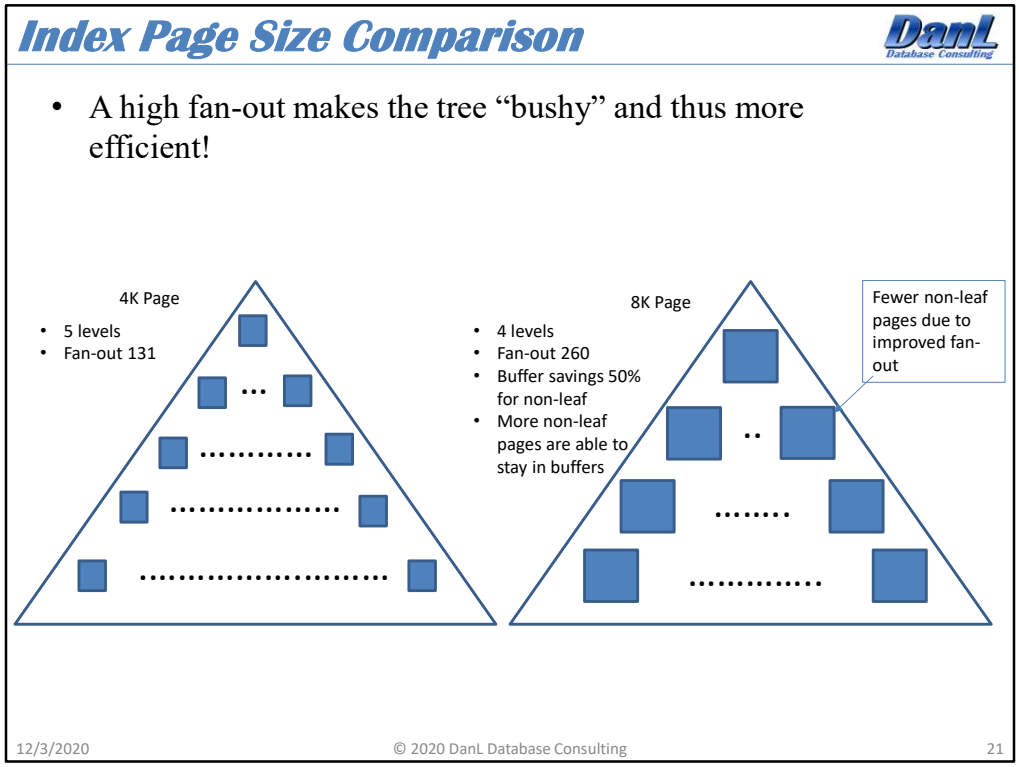


Index fan-out is defined as the number of children a node in a b-tree can point to. So, by having a larger page size, and also possibly compressing the index key, you can get a greater index fan-out. This has sort of a double effect, or maybe a quadruple effect with compression.

Let's look at one partition of one index. There are 137,160 active pages and 136,120 leaf pages. That means about 1,040 non-leaf pages, and a fan-out of 131. If we increase the page size to 8K we will cut the number of leaf pages in half, but also approximately increase the fan-out by a factor of 2 because the non-leaf pages are also larger. So, if I just use simple math 136,120/2 is 68,060 leaf pages. Now those non-leaf pages will be reduced by a factor of 4 to about 260, with a fan-out of 261.

It looks like compression will get us another 50% reduction in leaf pages to 34,030. I don't think we reduce non-leaf pages in this case.

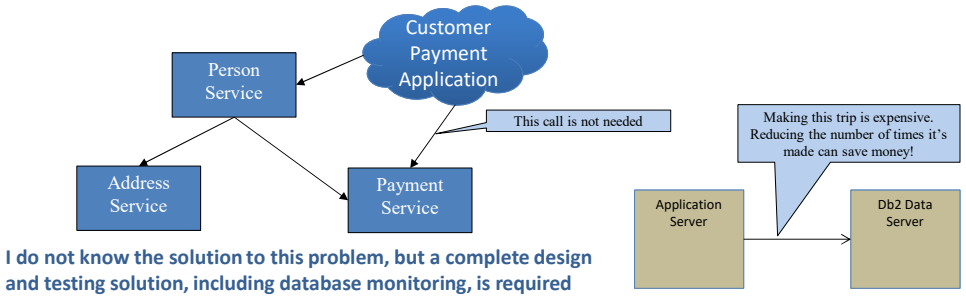
As far as the buffer pool go the compression does not save space in the buffer because the index pages are expanded in the pool. However, the large page size gets us the increased fan-out, which does reduce the amount of memory in the buffer pool required for non-leaf pages. This is due to the double effect I mentioned earlier with more entries per page and half the leaf pages for the non-leaf pages to point to. So, in this example we could possibly cut non-leaf buffer storage requirements in half, from 3GB to 1.5GB.



The potential savings is two fold. First there could be fewer I/O's because there are less pages to read in the larger page size index. Second, because of the improved fan-out the number of non-leaf pages can be drastically reduced and thus require less actual memory inside the buffer pool. There is also one less index level to traverse.

Service-Oriented Programming

- SOA, Agile Development, Automation or deployment and testing (e.g. Maven) dominate application development today
- It is quite common for services to be called too often in this environment



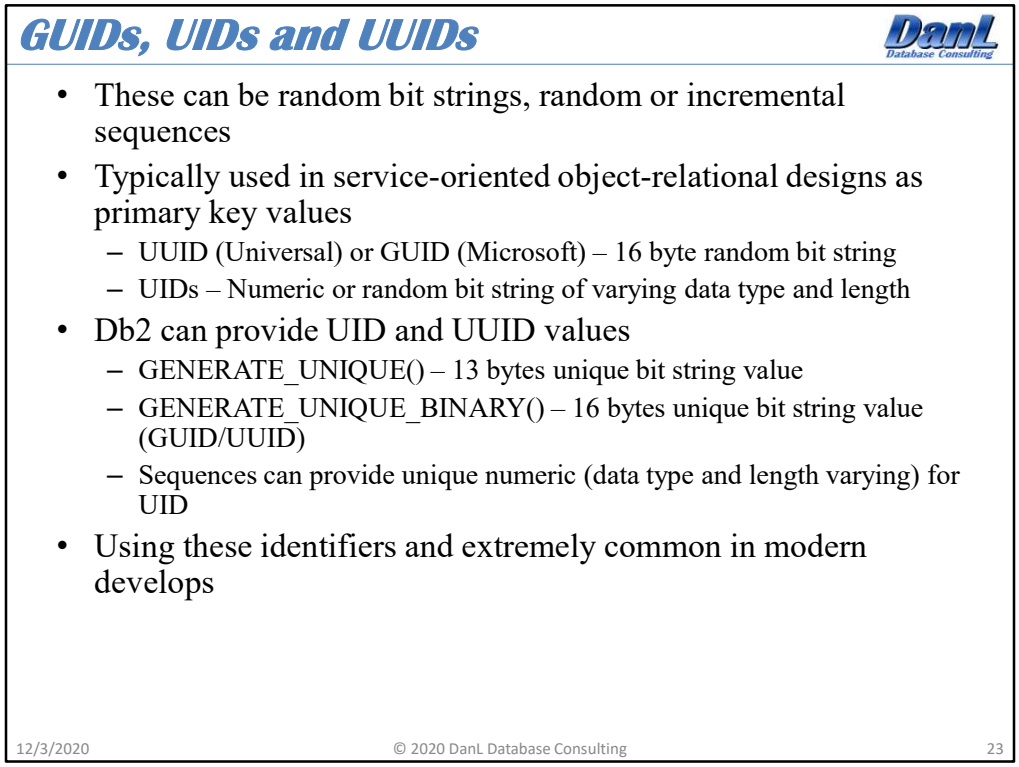
I do not know the solution to this problem, but a complete design and testing solution, including database monitoring, is required

12/3/2020 © 2020 DanL Database Consulting 22

Service oriented Agile development practices using frameworks and open software components are really important for cost savings related to application development. However, if the framework used and the development techniques are not uniquely controlled (and they often aren't), then this could lead to some sloppy design. It is important that there is a proof of concept, or at least accurate testing involving the database, to be sure that the database is being called the least number of times for a given service or transaction.

At one customer site each service included an optional "test" mode, which could be invoked via a global parameter. When in "test" mode the service would log the service name and entry and exit time each time the service was called. In that way we could track the relationship between a business transaction and the various service calls that support that transaction. The result of this policy resulted in a tremendous amount of machine time savings, improved transaction response times, and no major impact to application development time. This was due to the fact that we could catch redundant service calls during testing.

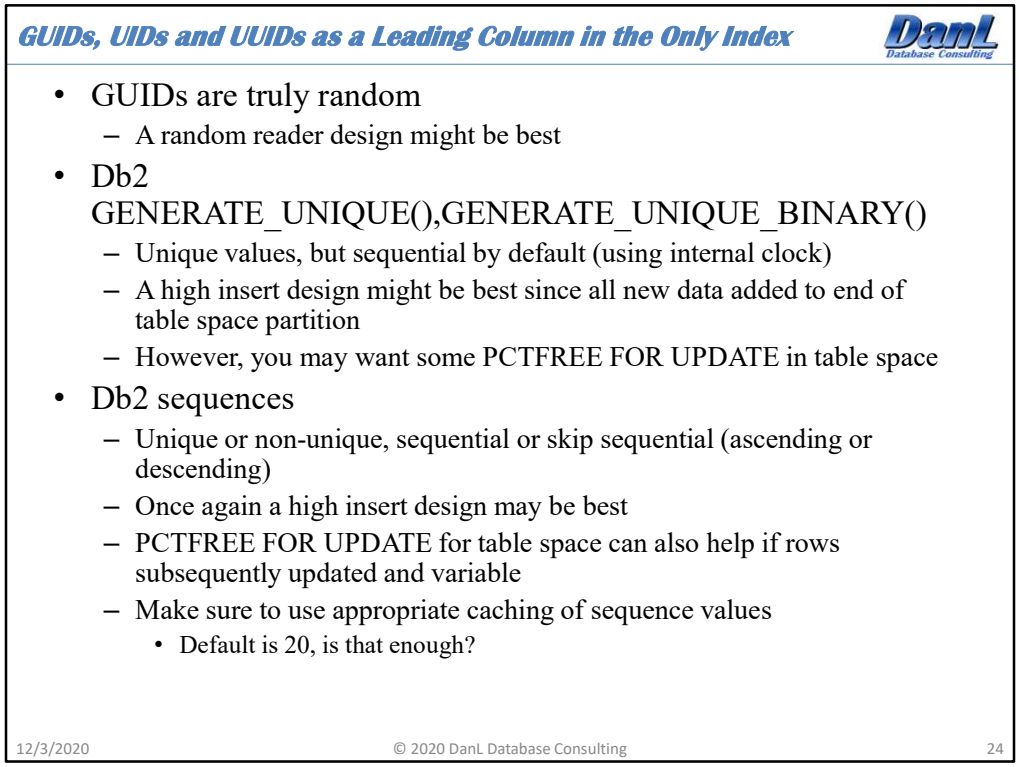
Calling the database server too often is the number one waste of machine resources that I have witnessed in all my years of working with Db2.



UUIDs are standard practice for many modern designs that incorporate an object-relational data stack. A standardized UUID is randomized based upon an internal time clock, but in practice it could be generated by any number of different algorithms. When UUIDs are introduced into a database design they can impact the decisions that are made relative to indexing, clustering and free space! Therefore, it is important to understand these IDs that are generated for a particular database design, and make appropriate accommodations. Perhaps it would be fruitful to address the need for natural keys for high volume tables, or possibly compound UUID keys for critical parent-child relationships.

It is possible to negotiate the natural-key "in" for a database design. Allow for clustered access based upon that natural key, and then cluster the rest of the tables by the UUID of that parent table, whether those keys be identifying or non-identifying. Remember this is something that is being done for high performance. If high performance is not an expected feature of the application then there is no need to fight the design team over primary keys.

If management tells you "we want the application to run as fast as possible" then your response should be "well great, buy an infinite number of machines". Ask for a reasonable expectation of performance, and if the use of UUIDs won't impede that expectation then relax, all is well. If there is fear of a performance bottleneck then special considerations should be made about the use of UUIDS, indexing, clustering, and free space!

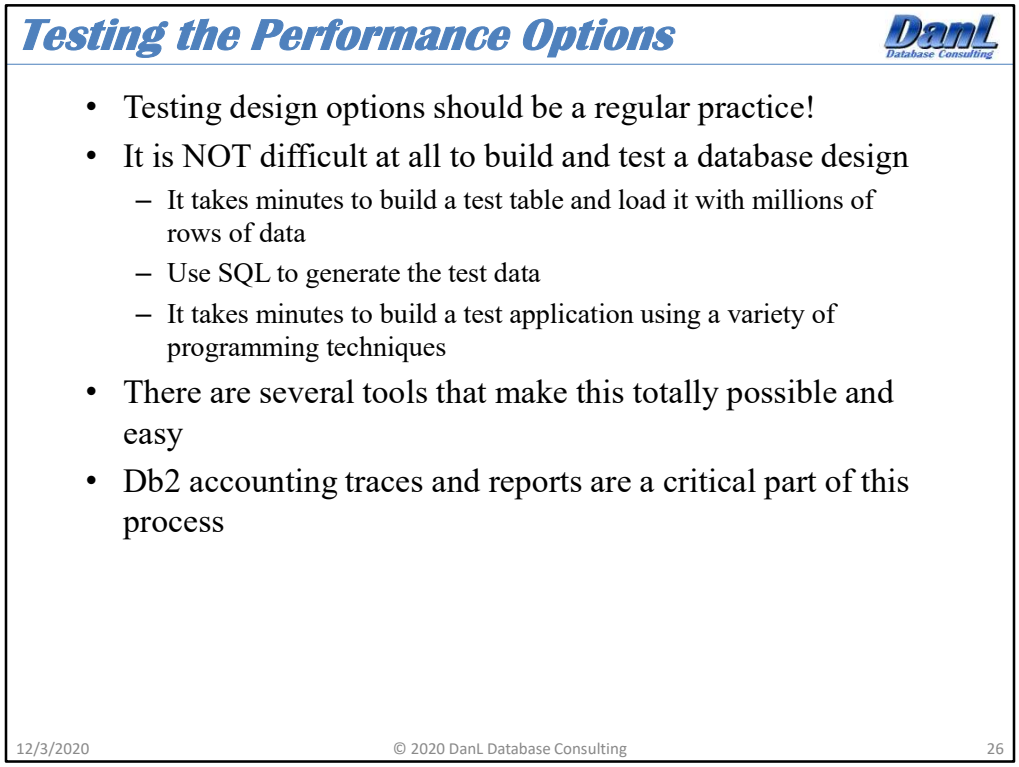


Clustering can be a challenge with a variety of generated unique id's as keys. If Db2 Generated values are used, whether they are GENERATE_UNIQUE() values or ascending sequence values then maintaining cluster based upon natural keys can be challenging without secondary indexes. However, you can create secondary indexes based upon a natural key, if one exists for the table, and cluster by that key. While the table may benefit from clustering, the indexes are another separate issue. It really depends upon the access patterns of the application queries, and which indexes they end up using.

A really common problem in Db2 is the use of sequence values without appropriate caching set. The default is 20, but if there is a high usage rate then that may be too small, especially in a data sharing environment. Setting ORDER and NO CACHE for sequence values is a massive trade-off for consistent numerical values and performance!

GUIDs, UIDs and UUIDs Considerations

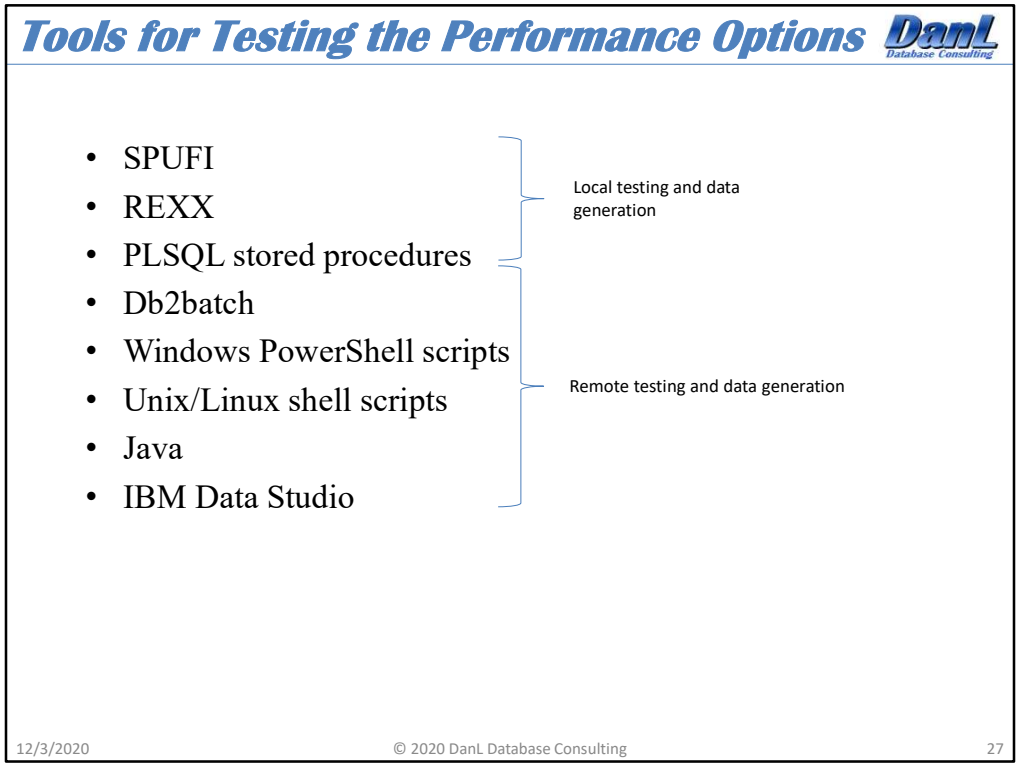
- GUIDs – Random
  - Allocate appropriate PCTFREE and FREEPAGE in primary key index
- Sequentially generated values
  - DB2 Sequences
  - GENERATE_UNIQUE(),GENERATE_UNIQUE_BINARY()
  - UIDs, UUIDs – if not random (again, lots of variances)
  - Little reason for any free space since primarily ascending
- As always, employ regular monitoring to understand the patterns associated with these or any indexes



After 30+ years there are few things that people can do or say during my day to day DBA activities that upset me. However, there is still one thing that can get me to walk out of a meeting! That one thing is the meeting in which people sit around and debate…." I think it will work like this", "That design will never perform properly", "you'll have to denormalize the initial design". Overthinking and trying to predict Db2 performance is a slippery slope, and so rather than "thinking" about Db2 performance why not have the database tell you how it will perform. Set up some proof of concept tests!

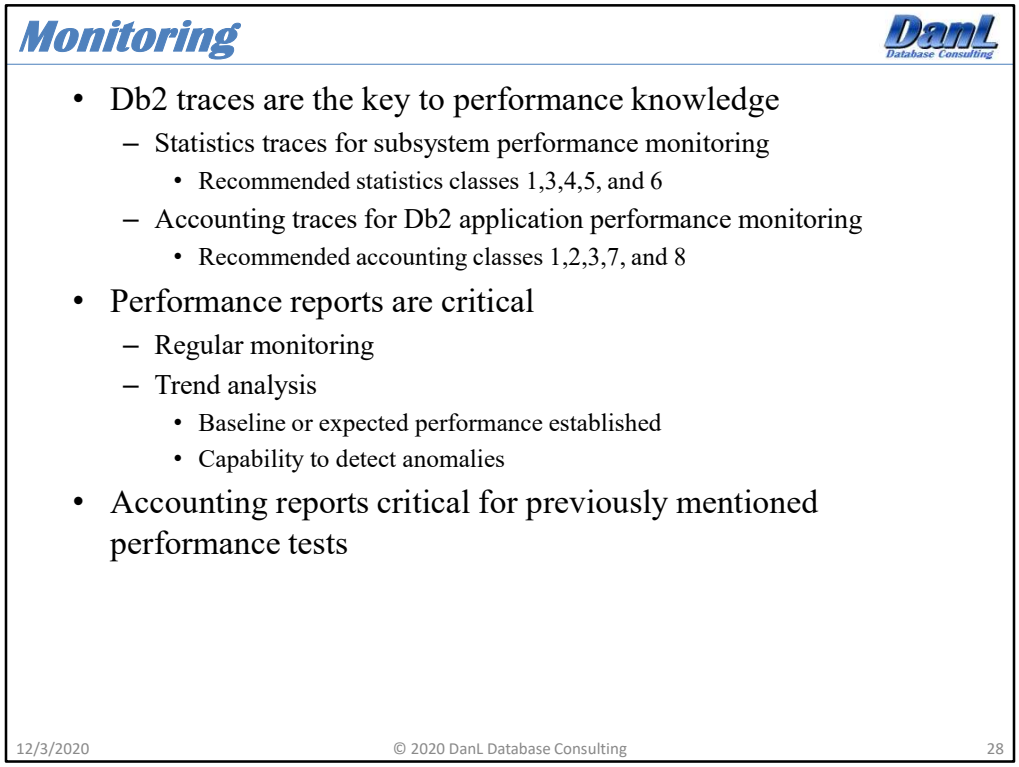
The accounting report plays an extremely valuable role in monitoring these tests. Contrary to what might be popular belief, it takes only a modest effort to set up some tables and populate them with test data. There are a variety of testing tools at your disposal; REXX, SPUFI, DSNTEP2, db2batch, and IBM Data Studio, to name a few. Run a test, produce an accounting report, change something, repeat. No more guessing as Db2 then tells you the best design choice for your application and database. I have done these dozens, if not more, times during my career.

IBM Data Studio uses a JDBC type 4 connection to Db2. This makes it easy to test JDBC connection properties, such as currentPackageSet, clientApplcompat, ClientUser, statementConcentrator, just to name a few.



Whatever the situation there is a no cost tool for the job. I have and still use every single one of these tools to test design and tuning ideas. There are plenty of examples to be found on the internet, and a decent combination of testing and monitoring will quickly point out the best design direction! I have been taking advantage of Windows PowerShell recently with great success! There is a wealth of information available online, and it is really powerful and easy to learn. Making a test table and generating data to load into the table using SQL and the RAND() function is fast and easy. Generating test data is just as easy. Consider a statement such as this:

```
SELECT CUSTNO FROM (SELECT CUSTNO, RAND() AS DANL FROM CUSTOMER_TABLE) AS TAB1
ORDER BY DANL FETCH FIRST 10000 ROWS ONLY WITH UR;
```

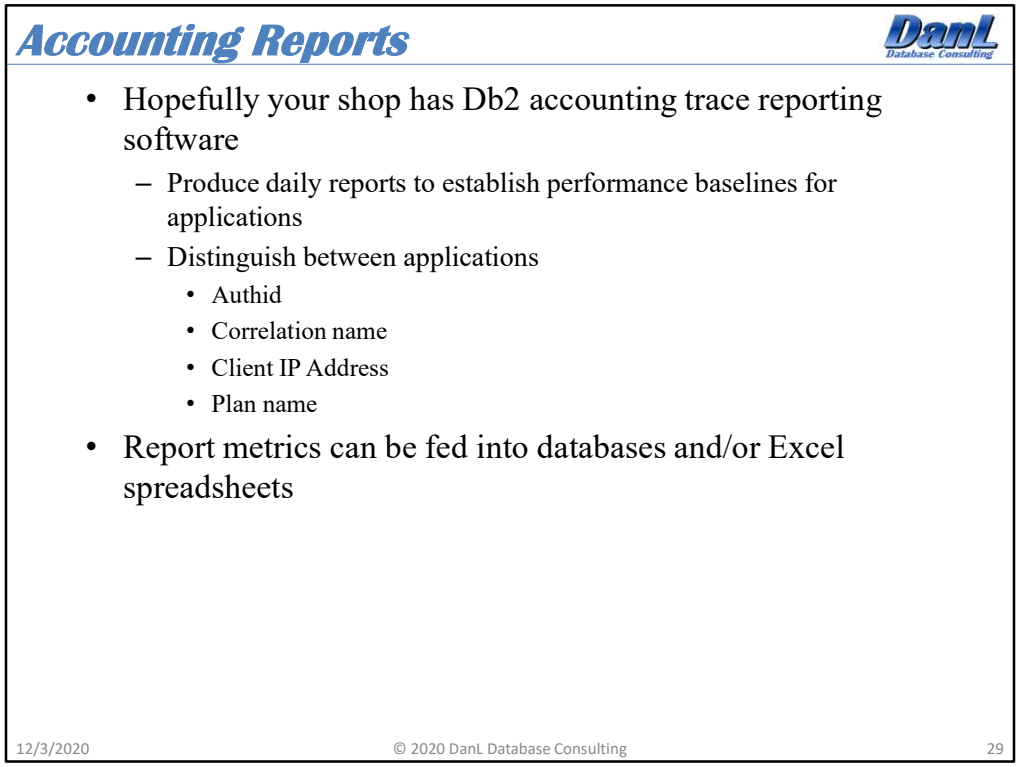


Db2 has an extensive built-in tracing facility that is extremely important when analyzing the impact of various settings, as well as overall performance. There are several types of Db2 traces.

- Statistics trace, which collects performance metrics at a subsystem level.
- Accounting trace, which collects thread level performance metrics and is really an essential key to understanding and improving application performance.
- Performance trace, which is not set by default and can be used to track specific threads or sets of threads at an extremely detailed level.
- Audit trace, which collects information about Db2 security controls and can be used to ensure that data access is allowed only for authorized purposes.
- Monitor trace, which enables attached monitor programs to access Db2 trace data through calls to the instrumentation facility interface (IFI).

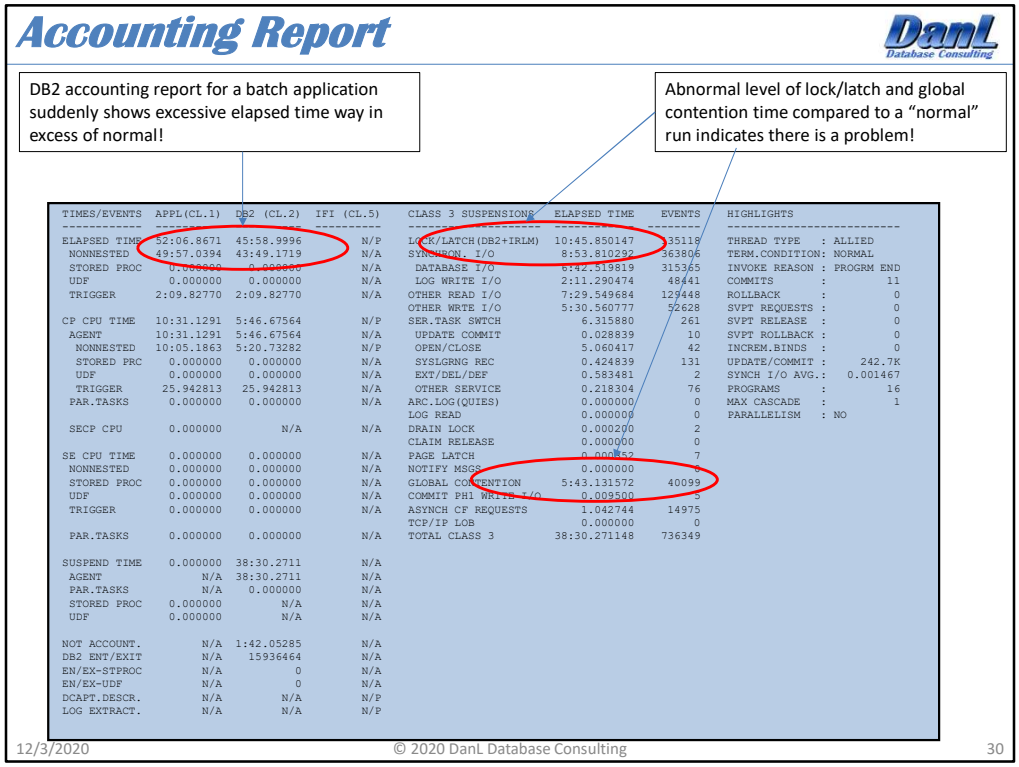
Traces can be set upon Db2 startup via the system installation parameters, or they can be started via Db2 commands. Typically Audit, Monitor, and Performance traces are set on demand and for specialized purposes. Statistics and accounting traces are typically set at Db2 startup. The most common settings are statistics classes 1,3,4,5, and 6 and accounting classes 1,2,3,7, and 8.

All of these traces are important and worthy of an understanding, but the focus of this article will be on the accounting trace. You can find descriptions of trace records in prefix.SDSNIVPD(DSNWMSGS).



The accounting trace collects thread level performance metrics and is really an essential key to understanding and improving application performance. The accounting trace records produced are typically directed towards system management facility (SMF) datasets and hopefully your friendly system programmer is externalizing these SMF records for Db2 subsystems into separate datasets for analysis. The accounting trace data these days is typically externalized upon the termination of each individual thread. So, you get one record per thread. This means that for large scale batch jobs, CICS transactions that use protected threads, distributed threads with the ACCUMAC subsystem parameter (this parameter determines whether DB2® accounting data is to be accumulated by the user for DDF and RRSAF threads) set at something other than NO, or high performance DBATs, you can get multiple transactions bundled together into the same accounting trace record. By default, Db2 only starts accounting trace class 1, which I believe is seriously not enough to accurately diagnose and track the performance of applications. My recommended accounting trace classes to set are 1,2,3,7, and 8.

Class 1 = Total elapsed time and CPU used by a thread while connected to Db2 at the plan level.
Class 2 = Total elapsed time and CPU used by a thread within Db2 at the plan level. This is a subset of class 1. The elapsed time is broken into suspension time and CPU time.
Class 3 = Total time waiting for resources in Db2 at the plan level. This is a subset of the class 2 time and is equal to the class 2 suspension time.
Class 7 and 8 = These are, respectively, the class 2 and 3 times divided between packages utilized during thread execution.

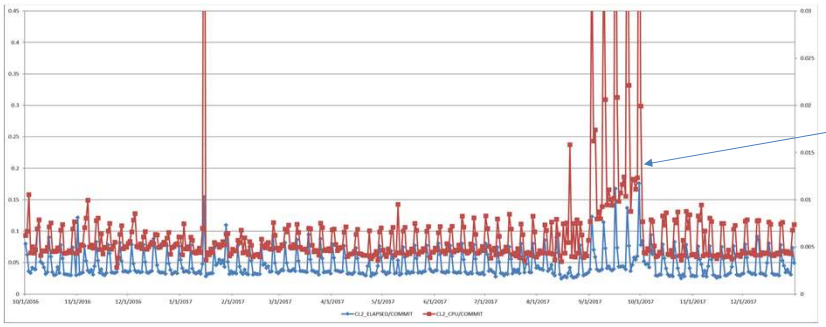


Here is an example of an accounting report that looked different than previous accounting reports for the application being monitored. The deviation from the norm manifested as excessive elapsed time, lock/latch time, and global contention time. It turns out that the cache size for a sequence object was set to the default of 20, and needed to be increased. The contention indicated in the report was actually against the Db2 system catalog as Db2 had to go there to get a next set of values for the sequence object. This was happening across a data sharing group and thus the global contention.

More on the accounting report can be found here:
https://www.db2expert.com/db2expert/the-importance-of-db2-for-z-os-accounting-traces-and-reports/

Accounting Database

- Db2 accounting trace data can be loaded into a database
- Database queries can then be used to track performance



Something went wrong here and needed to be investigated. Turns out some new untuned SQL was installed in the application.

12/3/2020 31

There are several tools that let you convert SMF accounting trace data into files that can be loaded into tables. To save on storage costs I typically load the data into Db2 for LUW tables on a department server, or my own workstation. I can then run regular queries that provide health checks for applications and place the results into Microsoft Excel workbooks to create graphs. Hint: managers like graphs!

Monitoring Indexes

- REORGs
  - Monitor indexes using real-time statistics catalog table SYSIBM.SYSINDEXSPACESTATS
  - Query example can be found at https://www.db2expert.com/db2expert/when-to-reorg-on-db2-for-z-os/
  - Key statistics
    - REORGLEAFFAR
    - EXTENTS
    - REORGMASSDELETE
    - REORGNUMLEVELS
- Separate indexes into their own buffer pool
  - Provides information about I/O activity
    - Separate from tables
    - In statistics and accounting reports
  - Place index of interest into its own buffer pool

Expect to REORG indexes more often than table spaces. You can run queries against the SYSINDEXSPACESTATS catalog table on a regular basis to determine if indexes require REORGs. Do not REORG table spaces as a habit if they don't need it. Just REORG the indexes as required.

If a particular application has a problem with I/O's, as indicated by high I/O wait times in an accounting report, it may be difficult to determine if the I/O problem is related to indexes or table spaces unless they are in separate buffer pools. Keeping indexes in their own buffer pool won't necessarily pinpoint the problem, but will give a clue as to whether it is an issue with table spaces or indexes.

References

- Db2 for z/OS index performance recommendations
  - https://www.db2expert.com/db2expert/db2-for-z-os-index-performance-recommendations/
- When to REORG on Db2 for z/OS
  - Includes queries to monitor indexes for REORGs
  - https://www.db2expert.com/db2expert/when-to-reorg-on-db2-for-z-os/
- Db2 for z/OS accounting traces and reports
  - https://www.db2expert.com/db2expert/the-importance-of-db2-for-z-os-accounting-traces-and-reports/
- INDEX COMPRESSION in Db2 Z, A recap and overview! By Brian Laube
  - Direct link will not work as you need an IDUG login (free)
  - Once logged in
    - https://www.idug.org/browse/blogs/blogviewer?blogkey=4b01dbe2-a8b6-4ae3-8fb4-d4c1411c402c
  - Or, log in and search on "Brian Laube Compression"

Dan Luksetich is a Db2 DBA consultant. He works as a DBA, application architect, presenter, author, and teacher. Dan has been in the information technology business for over 35 years, and has worked with Db2 for over 30 years. He has been an application programmer, Db2 system programmer, Db2 DBA, and Db2 application architect. His experience includes major implementations on z/OS, AIX, i Series, Windows, and Linux environments. His industry experience includes retail, banking, fraud detection, analytics, government, and utility. He specializes in highly available, high-volume transaction processing against very large database systems.

Dan's experience includes, but is not limited to:

- Application design and architecture
- Business analytics
- SQL consulting and education
- SQL coding and application programming
- Database administration
- Complex SQL
- SQL tuning
- Db2 performance audits
- Replication
- Disaster recovery
- Stored procedures, user-defined functions, and triggers
- Db2 REST services

Dan likes beer and is a Certified Cicerone!