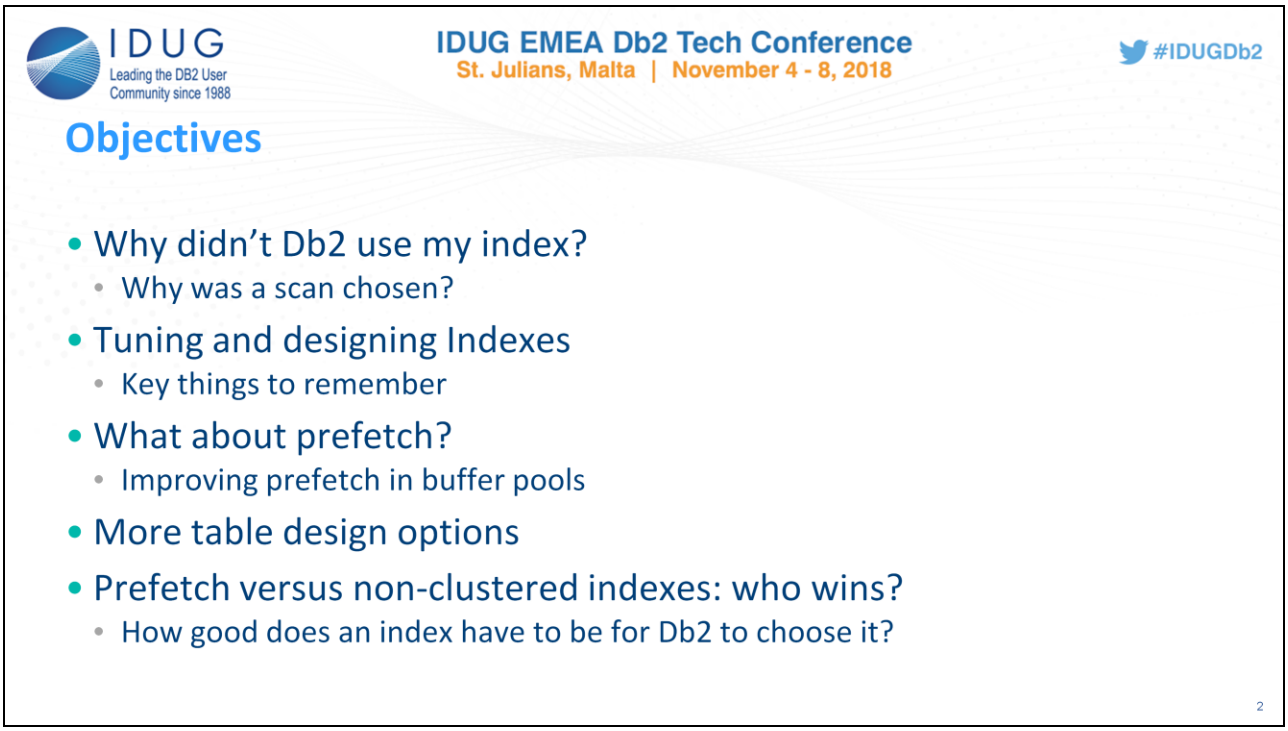
Presentation Abstract

From the beginning of DB2, application performance has always been a key concern. There will always be more developers than DBAs, and even as hardware cost go down, people costs have risen to even higher levels. Indexes are often added to fix problems, but many don't help at all. The presentation looks at techniques to judge which indexes are good and which are evil, and techniques to improve application I/O.



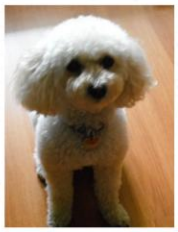
Frame of Reference

I have worked with DB2 since 1985. I was pushed into consulting in 1992, and I started with Db2 for LUW in 1993.

Since 1997, tuning vendor applications, first Peoplesoft, then Siebel and other industry applications. Now I help customers with SAP, Sabrix, Vertex, WS Commerce, Sterling Commerce, etc. I am still tuning in-house applications.

I have spoke at IDUG NA since 1989, EMEA since 1995.

Why didn't Db2 use my index?



- While a lot of index use is simple
  - Get a row via a primary key
  - Matching indexes to SQL is mostly straightforward for simple queries
- But other indexes might not be helpful:
  - Index columns do not match the predicates of Where clauses
  - Order of index columns not ideal
  - SQL is complex with many range predicates
- Bottom line:
  - Some indexes do not save enough I/O to be considered useful by Db2's optimizer

Why was a scan chosen instead of my index?

- Short answer: cost
  - Db2's optimizer computes the cost of many access paths
  - Chooses the lowest cost based on:
    - I/O for both data and indexes
    - CPU for sorts and other CPU operations
- Other notes:
  - Db2 LUW uses CPU speed in its costing algorithm

Let's make a fair comparison

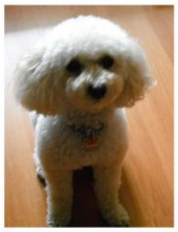
- We love indexes and their use
  - We want the best index design possible for our applications
  - Index design is always a popular topic at IDUG
    - Sessions are usually well attended
- On the other hand, scans and prefetch are not what we want
  - Perceived as a failure to have enough indexes
  - We can also tune to improve prefetch
  - There are other table design options that combine both concepts
- Let's discuss what we need to compare optimal indexes to a well-tuned DB with good prefetching capabilities

5

We always want data retrieval to be done by indexes. While there is some art, meaning knowledge and experience, to designing the best indexes, there are also tools such as the Design Advisor to assist in this process.

Changing indexes to improve performance

- Many people feel comfortable adding indexes
  - Some people get really comfortable
- The same people are reluctant to remove indexes
  - "The indexes might not help now, but they might in the future"
  - "The vendors tell me they know what they are doing"
  - "Elephants and mosquitoes"
  - Indexes help with retrieval, but slow data change processes
- On behalf of my children, I thank them all
  - Many SQL performance problems can be solved by fixing the indexes
  - Dropping poor indexes can speed up applications



Index Tune-up Guidelines

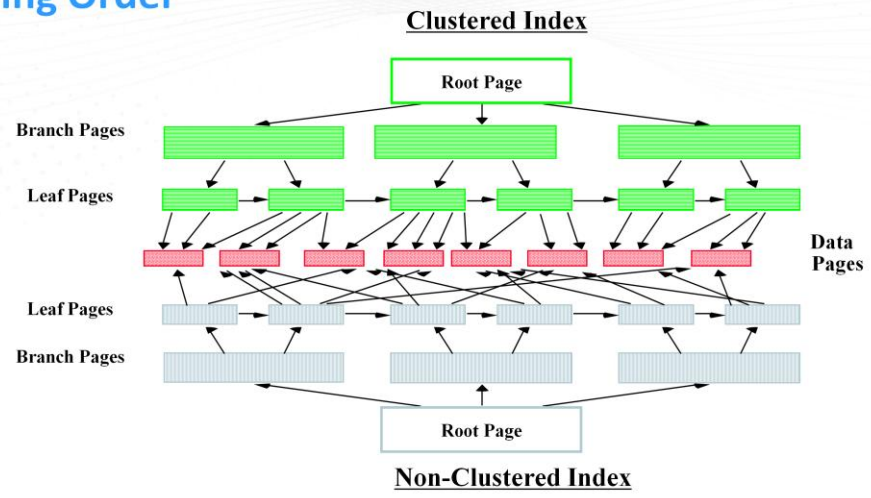
1. Do not change the primary index
2. Plan for little index-only access
3. Check clustering index placement
4. Remove redundant indexes
5. Remove unnecessary index columns
6. Add index columns if needed
7. Add indexes carefully when necessary

This list represents general guidelines for tuning indexes of an existing application. Each point will be discussed in the following pages.

Clustering

- Clustering choice is often not intentionally made!
  - Clustering often ends up on primary key by default or by mistake
- Put clustering index on the most time-sensitive process
  - Valuable low cardinality index
  - Highest use with range predicates
  - To avoid sorts
  - To uniformly spread INSERTs

Clustering Order



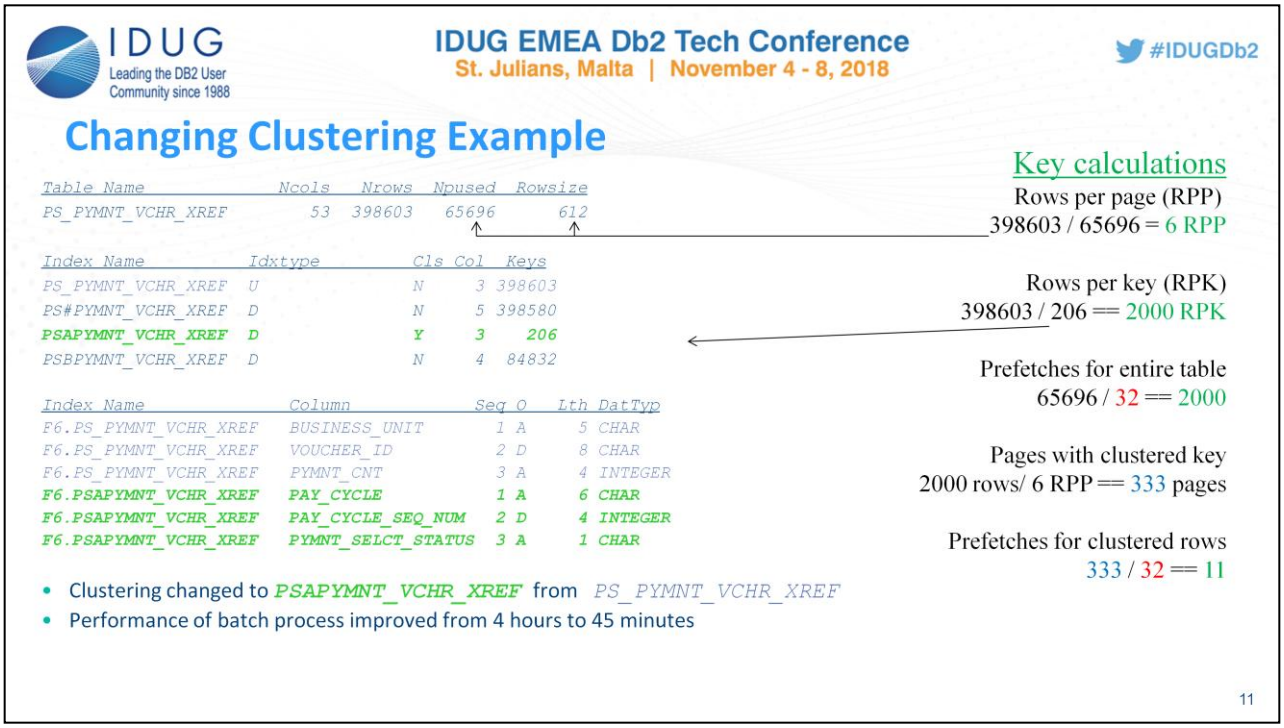
The difference in using a clustering index is actually how the table space pages are processed. If there has been no disorganization of the table space, the data in the table will be stored in the order of the index. For duplicate key values or range predicates, these rows will be found adjacent to each other. The result is that far fewer data pages are processed.

The Informix optimizer also takes into account the use of the clustering index. For an index in non-clustering order, Informix assumes that each row will be found on a different page. Rows retrieved via a clustered index will perform substantially fewer table space I/Os. Informix will use the catalog statistics to compute the number of pages to be retrieved based on the number of rows per page.

This substantial saving dictates that the person doing the physical design choose the clustering index carefully. Clustering indexes will help when cardinality is low, or when range predicates such as BETWEEN, LIKE, or < are used.

Cluster Facts

- Access path calculation
  - Non-clustered index: one I/O per row (worse case)
  - Clustered index: compute rows per page (RPP)
    - Compute rows per prefetch operation (RPP *NumPages)
    - Clustered indexes are clearly the best choice for multi-row answer sets
- Low cardinality columns may still help with data retrieval
  - If the index is the clustering index
  - Data may be skewed
- Multi-dimensional clustering (MDC) is a viable alternative for:
  - Skewed data
  - Some data inserted in time sequence benefits from ITC (Insert Time Cluster)
    - Roll-in roll-out is very easy for time-based data



In this example, voucher processing was following the order of the PSAPYMNT index, but the "_" index was the clustering index. Changing cluster to the "A" index reduced the run time of this critical job from 4 hours to 45 minutes.

The calculations are from a real customer situation, where the application was fairly new. Calculating the Rows per Page, Rows Per Key and the Number of Prefetches to read a table are good things to know when tuning.

This example uses an arbitrary prefetch quantity of 32 in the calculations. Newer versions of DB2 LUW would likely use 64 or 128, and this would make the case stronger for changing the clustering. The data grew substantially over time, making this decision even more important.

Order of Index Columns

- Example: a stock pricing table containing the closing price for 300 stock symbols:
  - Data will be stored daily for 10 years
  - Two columns in index:
    - Date
    - Stock symbol
- Which column should be first in the index?
  - Retrieval is for a single stock symbol

This example can from a small internet stock application. For 300 stock symbols, Some people felt that it was better to place the daily stock price first in the index if 10 years of history were kept.

Order (cont'd)

- Date was specified originally as the first column
  - Had the higher cardinality
    - 3650 for 10 years' history
- Dates are often used with a range
  - Ordering is off for the second column
- Stock symbol was better answer
  - Lower cardinality, but always used with an equal predicate
- The Bottom Line: Dates should never be the first index column
  - Yes, you provide an example to the contrary…

But, because dates are almost always specified as a range, more index scanning was required to find the required symbol. Changing the order in the index provided the needed performance gain.

Tuning vendor indexes

- Some comments from vendors:
  1. "OUR indexes are well designed" and "OUR queries return only 1 row"
  2. "You might be in big trouble for that index you dropped when you need it a year from now"
  3. Conversely : "Independent tuners can add value for customers"
- My investigations:
  - Most vendor indexes are well designed for delivered code
  - Few customers implement all search fields
  - Almost all customers have customized functionality that requires index changes
- How long does it take to add an index (or put it back if deleted)?

Comments on Indexes from Application Vendors

- Vendors try for uniformity across DBMS products
- Few vendors understand Db2 clustering
- No MDC tables
- No include columns in delivered indexes
- For example, in 6 years with SAP
  - Added about 200 indexes
  - A few tables ended with up to 8 indexes, rest less

In some respects, vendors do not support the advanced features of DB2 LUW. Include columns, clustering and MDC are not exploited, and if you define the indexes through vendor tools, these cannot be specified.

Another great source of index information

- The Design Advisor is a great tool
  - Free and part of DB2
- Go a step further – review Scott Hayes' 2016 presentation C03
  - In Part 3 from 2016, Scott uses the advise tables to turn off and on recommended indexes to see the individual impact of indexes
  - This is the best analysis anywhere

16

The design advisor continues to be one of the best ways to improve Db2 performance through the automated evaluation of indexing and design alternatives. If it is used in single statement mode, you may have "created a monster" as you might find that some people end up with one index per SQL statement per table.

Scott's presentation shows how to take the analysis a few steps further when evaluating indexes for workloads.

What about prefetch?

- Concepts of logical and physical I/O
- Logical request for data is an application's request for data
  - Rows selected, rows fetched
- If data is not in BP, physical I/O involves retrieval from disk
  - Causes delays & consumes CPU
  - Logical I/O satisfied from the buffer pool requires no physical I/O
  - Shows as Rows read
- And mainly, why separating random and sequential I/O is essential to good performance
  - And how to do it!

The fact that DB2 allows for the definition of buffer pools leads one to think that splitting DB2 objects into separate buffer pools would be a good thing to do. This is a true statement, and we will discuss buffer pool allocation strategies.

In DB2, there are 2 basic types of I/O: logical and physical. Physical I/O is the traditional type that is viewed as bad: I/O requests are made to devices, disks rotate, and eventually data is returned to the application. An I/O causes delays waiting for it to complete and consumes CPU resources.

Logical I/Os are simply application requests for data. Through the use of areas of memory, called buffer pools, DB2 can substantially reduce CPU, I/O, and elapsed time. This is reflected by reduced resource consumption and improved user productivity.

It is desirable for a logical I/O to not require a physical I/O. If the data resides in the buffer pool(s), the request is much cheaper than waiting for physical I/O to complete. Updates to data are made in the buffer pools, and these changes are written back to disk when threshold values are reached or at system checkpoint time. The buffer pools in central storage are also called the virtual buffer pools.

Buffer Pool I/O

- Concepts of logical and physical I/O
- Logical request for data is an application's request for data
  - Rows selected, rows fetched
- If data is not in BP, physical I/O involves retrieval from disk
  - Causes delays & consumes CPU
  - Logical I/O satisfied from the buffer pool requires no physical I/O
  - Shows as Rows read
- And mainly, why separating random and sequential I/O is essential to good performance
  - And how to do it!

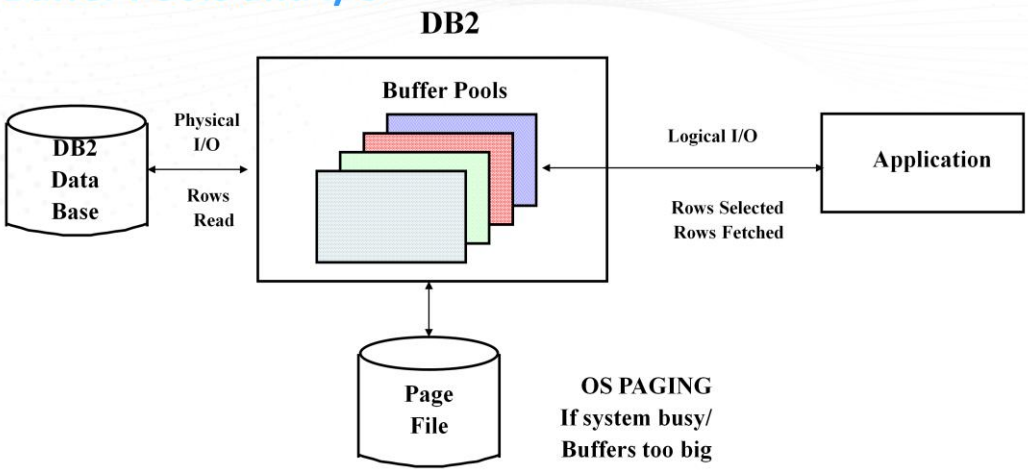
The fact that DB2 allows for the definition of buffer pools leads one to think that splitting DB2 objects into separate buffer pools would be a good thing to do. This is a true statement, and we will discuss buffer pool allocation strategies.

In DB2, there are 2 basic types of I/O: logical and physical. Physical I/O is the traditional type that is viewed as bad: I/O requests are made to devices, disks rotate, and eventually data is returned to the application. An I/O causes delays waiting for it to complete and consumes CPU resources.

Logical I/Os are simply application requests for data. Through the use of areas of memory, called buffer pools, DB2 can substantially reduce CPU, I/O, and elapsed time. This is reflected by reduced resource consumption and improved user productivity.

It is desirable for a logical I/O to not require a physical I/O. If the data resides in the buffer pool(s), the request is much cheaper than waiting for physical I/O to complete. Updates to data are made in the buffer pools, and these changes are written back to disk when threshold values are reached or at system checkpoint time. The buffer pools in central storage are also called the virtual buffer pools.

Buffer Pools and I/O



Buffer pool tuning is a major method for tuning applications. It is generally done in the production environment where the workload to be tuned resides. This diagram shows where logical I/O and physical I/O are performed.

If there is insufficient real memory to back up the specifications for buffer pools, OS paging will occur. This means that the entire system performance will be negatively affected. If this has occurred due to DB2 buffer allocation, the buffer pools should be reduced immediately. If you did this without telling your system administrator, he will NOT like you very much!

Types of Physical I/O

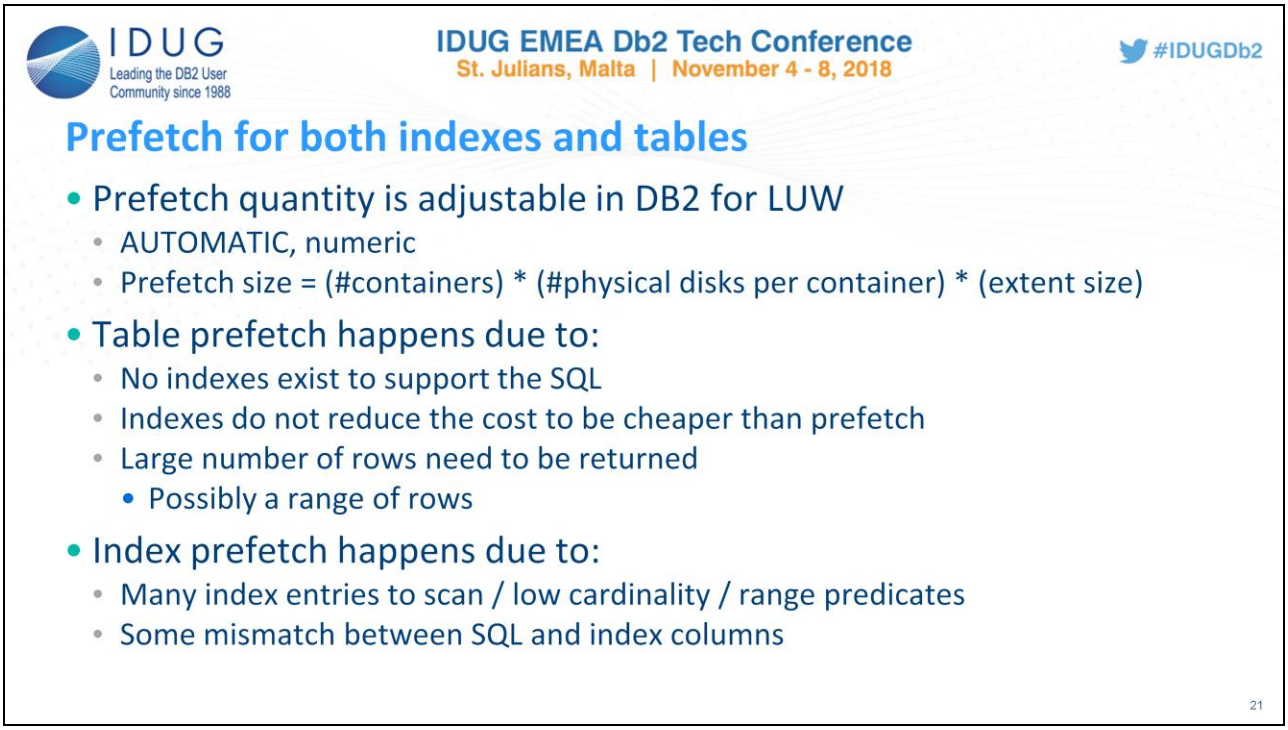
- Random or "synchronous"
  - Random I/O means an available index was used
- Prefetch
  - "Read-ahead" capability for data and indexes
    - Reduces waits caused by reading ahead from disk
  - Also called "asynchronous" or "sequential" I/O. Types:
    - Pure sequential
    - List
    - Smart data/index (dynamic & readahead (10.1))
      - Dynamic (sequential detection) for highly sequential patterns during execution
      - Readahead when data is badly clustered

Physical I/O includes many additional things such as rotational delay for the disk drive and contention for the device and channels. A good access time for a single page from disk is 6 milliseconds. A read satisfied from the buffer pool is measured in microseconds, or at least 100 times faster.

There are two major types of I/O performed by DB2. The first type is random I/O where a given data request uses an index to find the appropriate page and retrieve the data required. Random I/O is efficient and is even better when no physical I/O is needed due to the data being in the buffer pool.

The second type of I/O is sequential I/O. As its name implies, sequential I/O requires that all or part of a table space or index be read sequentially to find the data required by an application. This usually means a lot of I/O and processing by DB2 to find the result requested. Sequential I/O can lower the possibility of finding a page in the buffer for the next request. Prefetch I/O takes longer than random I/O, but it reads many more pages per I/O.

Pure sequential prefetch means that the optimizer at bind time has chosen to use prefetch. List prefetch is like skip sequential processing: a list of Row IDs is built from one or more indexes and sorted before the data is retrieved in the order it is stored on disk. Dynamic prefetch, also called smart prefetch or sequential detection, is a run time decision by DB2 in response to a sequential pattern in the data. If the pattern changes to a point where sequential prefetch is not helping, it is turned off and possibly back on later.



There is no metric that shows the actual prefetch size if automatic is specified, but it works quite well.

A unique index with an equal predicate uses the least amount of I/O for a traditional query: one I/O per level of the index followed by the reading of a single table row. Other very efficient ways to retrieve data include index only access and even single fetch index access. If all data access could be so lovely!

But users and business requirements often ask more of the database. Often many rows need to be returned, or scanned to find the answer set, and the result is(much) more I/O.

The Pros and Cons of Prefetch

- Positive side of prefetch is that the worst case is made much better
  - V1R1 of DB2 for MVS did 1 I/O for each page synchronously
    - Very slow and costly
  - Now, wait time is reduced and number of pages per I/O is much higher
- Negative side of prefetch is that a lot of I/O is required
  - As calculated by Db2 optimizer
  - Reduced wait time but a lot of data is scanned
  - Subsecond transaction response is unlikely
- (That's why you are in a presentation comparing indexes to prefetch)

22

In the very first days of Db2, the early version of the optimizer had the choice of using an index or reading the entire table. As noted earlier, the introduction of prefetch helped a lot. New methods of using prefetch where introduced to further exploit the benefits of asynchronous I/O. Prefetch is all about reducing the wait time to retrieve large amounts of data from disk.

To know that a lot of data is required is pretty much the entire downside of prefetch. You will be doing a lot of processing.

Methods Considered for BP Tuning

- Increase size of BP
  - Arguably the most obvious and effective way to reduce I/O
  - In LUW, memory is cheap and often sized larger than z/OS
  - Memory can be deployed such that its benefit is greatly reduced

1. Separate by type of I/O
   - High asynchronous I/O can force high-use pages from BP
   - Need high use pages to remain resident – presumably accessed via index
2. Separate tables from indexes
   - Indexes are expected to be smaller than tables
3. Separate by volume
   - High-use objects are isolated from other objects in the hope of increasing page residency

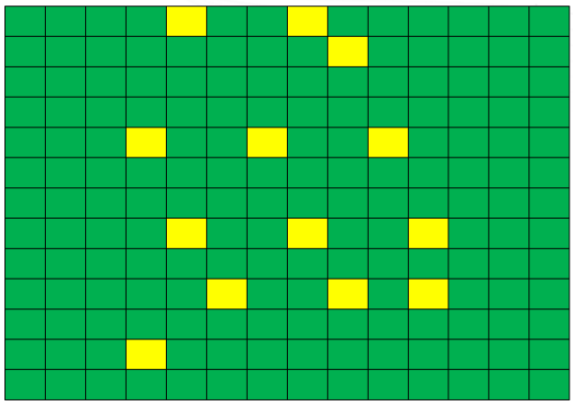
- **Of the 3 ways to separate objects, which is the most effective?**

To obtain the best tuning results, it is important to tune for the peak periods for application use. Large buffer pool sizes combined with proper allocation of objects to buffer pools can make dramatic improvements to application performance.

The immediate benefits of buffer pool tuning are improved on-line response times and batch run times realized by users of the DB2 system, improved user productivity, greater system throughput, reduced CPU utilization, reduced I/O workloads, and the optimization of DB2 memory resources.

Buffer pool activity illustration

- Individual or groups of pages can be read from disk
  - For random I/O, pages will fill buffer
  - Least recently used (LRU) algorithm used to write over oldest unlocked pages
- A bigger BP means more pages in buffer
  - Some highly used pages may remain resident for long periods



24

DB2 externalizes old pages according to the changed pages threshold. IO cleaners perform this operation asynchronously.

Pages that have been written back to disk can be stolen, but they can also be read again if they have not been replaced by other activity.

"Accentuate the positive"

- Random I/O is positive
  - Means indexes are used
  - Keep high use pages in BP
- Make the BP large enough to avoid physical I/O
- NOTE: Know your real memory size
  - Monitor available storage with tools like vmstat or nmon
  - Do not allocate memory without speaking to your system admin

25

The idea here is to use the available memory on the machine. Memory is very cheap on x86 machines, and comparatively cheap to mainframes even on AIX machines. Many machines have unallocated memory that could help to improve DB2 performance.

"Eliminate the negative"

- We want maximum residency for high use pages
- Let's remove I/O or objects that negatively impact page residency
- Two primary methods:
  1. Separate random from sequential I/O
  2. Separate objects that negatively impact page residency

26

The biggest negative impact comes from sequential read operations that overlay pages within the buffer pool.

Eliminate Sequential From Random I/O

- Sequential I/O involves reading a large number of pages to find data that qualifies
  - Can push many high-use random pages out of the buffer pool
- In the past, it was prudent to use separate buffer pools for randomly and sequentially accessed objects
- Use block based buffer pools
  - Can effectively separate within a buffer pool using the block based area

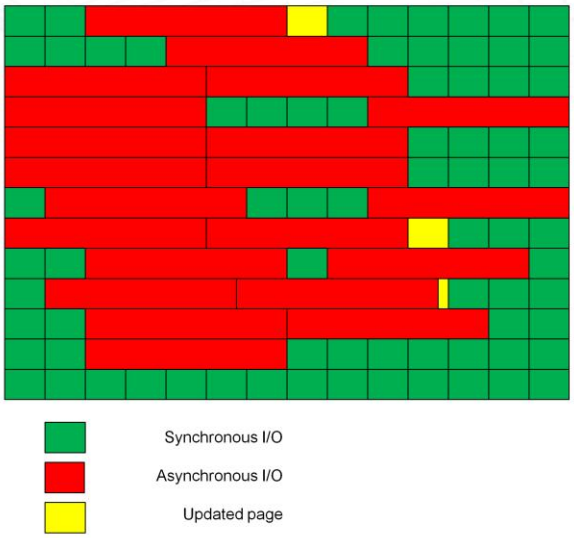
Random I/O is efficient and is even better when no physical I/O is required to satisfy the request. A large buffer pool will allow high-use pages to remain in the buffer pool.

Sequential I/O lowers the possibility of finding a page in the buffer for another request. Even if a large buffer size is used, there is little chance of a page still being resident. Separating sequentially accessed objects into a buffer pool away from randomly accessed objects will improve the performance of the random objects.

Sequential objects should not be expected to perform well regardless of the buffer pool size. A pool of 1,000 to 3,000 buffers should be adequate depending on the number of objects and how heavy the workload is. A larger number may be required.

Effect of sequential I/O

- DB2 uses prefetch to avoid waits for I/O
  - Each prefetch operation can overlay 32+ pages in BP
  - Hit ratio for randomly accessed pages can be drastically reduced
  - Must be reread from disk
- Sequentially read pages are rarely used again
  - Scans simply replace oldest pages



Block-based buffer pools

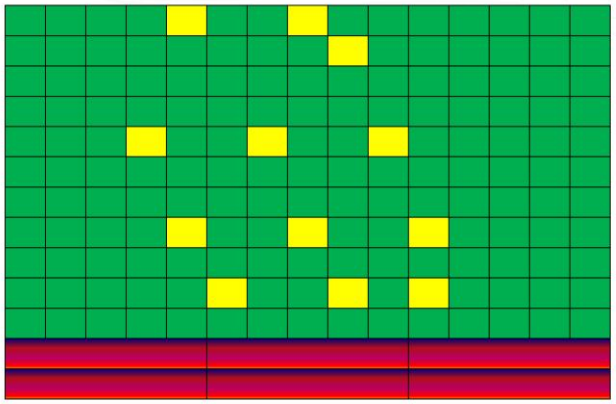
- A block-based buffer pool consists of
  - Page area - non-sequential (random) workloads
  - Block area - a specified number of contiguous pages
- If block based buffer pool is available
  - Significantly improves residency of non-prefetched pages
- CREATE and ALTER BUFFERPOOL
  - NUMBLOCKPAGES <number-of-pages>
    - Number of block pages must not > 98 % of BP pages
    - Specifying value 0 disables block I/O
    - NUMBLOCKPAGES should be multiple of BLOCKSIZE
  - BLOCKSIZE <number-of-pages>
    - 2 <= Block size <=256. For SAP use 2.
    - Number of pages read from disk in a block I/O

One of the biggest, yet not often spoken about, enhancements in DB2 V8 are block-based buffer pools. The block-based area within a BP allows sequential operations to be separated from random I/O. Just specify a block-based area, and this will happen automatically.

The Blocksize should be equal to the Extent size for the table spaces using your buffer pool.

Effect of sequential I/O with block-based BP

- Prefetch occurs in block-based area
  - Sequentially read pages are used and written over
  - Hit ratio for randomly accessed pages is perserved
- Block area does not have to be large to give good results
  - 1-3 percent of a large pool is enough



Rethinking conventional wisdom

- Traditional advice (especially for z/OS):
  - Separate randomly and sequentially accessed objects
  - "RAMOS" and "SAMOS"
  - Measure I/O characteristics carefully and place objects in correct BP
- **New thinking**:
  - Block area separates random and sequential I/O ***automatically***
  - *Determined by DB2* where each I/O should be done: page or block area
  - Even for a single object, the block area is used
- **Separate BPs for type of access is not needed in DB2 LUW provided you define a block area**
- **A single block-based buffer pool often provides reasonable performance**

31

RAMOS – random access mostly

SAMOS – sequential access mostly

The need to separate objects based on their I/O characteristics is a moot point if a block based area is used.

Block-Based Calculations

- Block I/Os are in block-based area
  - Vector I/Os are sequential I/O in page area of BP
  - Most sequential operations should happen in block area

```
Asynchronous data read requests          = 2994624
Asynchronous index read requests         = 713859
Unread prefetch pages                    = 3442482
Vectored IOs                             = 294962
Pages from vectored IOs                  = 2047922
Block IOs                                = 2793441
Pages from block IOs                     = 82798654
```

- Pages per block I/O
  - `82798654 / 2793441 = 29.64 –(Blocksize 32)`
- Percent of block I/O of sequential I/O
  - `(2793441 / 3088403) * 100 = 90.4%`
- Percent of prefetch pages from block I/O
  - `(82798654 / 84846576) * 100 = 97.6%`

These three calculations show the efficiency of the block-based area and its ability to handle the amount of sequential activity. The efficiency of the area appears good, and the number of pages per block I/O is approaching 32, which is the blocksize and the prefetch quantity.

Ideally, 100% of the pages come from the block-based area. Having 97.6% of the pages happen there seems quite good.

Block-based BP: my thoughts

- Virtually every DB2 object has some form of sequential access
- Block area gives automatic separation of random and sequential I/O
- Every BP should have a block area defined
  - Only 1 – 3 % of pool size: a little is enough
  - Even if no sequential access, a small block area avoids a problem
  - No other action required to separate synchronous / asynchronous I/O
- Measurements from BP snapshot show effectiveness:
  - Block I/O – 90%+ show be in block area
  - Vector I/O – sequential I/O in page area. Should be zero or <10%
  - If Block I/O is zero with block area – the definition is not correct
    - Match block size to extent size – SAP uses 2
- High sequential access may indicate indexes are needed

33

A block based area is like cheap insurance to ensure more randomly accessed pages remain in the buffer pool. If the block based area is not large enough to hold all aynchronous activity, it spills into the the page area as vector I/O.

Of course, high sequential activity could point to the need for indexes to support certain queries.

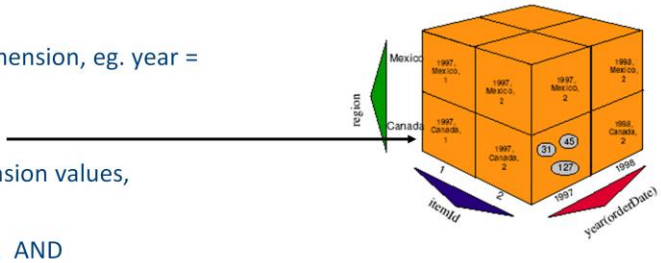
More Table Design Options: MDC and ITC (10.1) Tables

- Multidimensional (MDC) and insert time clustering (ITC) tables
  - Similar technology – use blocks for similar data
  - Create MDC via ORGANIZE BY (columns)
- Create ITC tables by specifying ORGANIZE BY INSERT TIME
  - Data clustered via a virtual column that clusters rows inserted at a similar time
- Data is stored within blocks
  - Blocks become free from batch deletes (rollouts)
  - Blocks can be read in a single I/O

I look upon ITCs as a good extension of MDC technology. The main concept is to group data into extents that can easily be retrieved via prefetch.

MDC Concepts

- Dimension
  - "Block" index column
  - eg, year, region, itemId
- Slice
  - Key value in one dimension, eg. year = 1997
- Cell
  - Unique set of dimension values,
  - eg, year = 1997,
  - region = Canada, AND
  - itemId = 1



Conceptually, MDC indexes are made up of dimensions, slices, and cells. These are translated into blocks and extents when they are physically stored.

One of the most exciting features in DB2 for LUW v8 was the introduction of multi-dimensional clustering (MDC). MDC indexes were designed to provide the benefits of clustering across each column within a multi-column index.They are introduced as a feature to help data warehouse applications, but they will also help OLTP applications as well.

Several new concepts and terminology are introduced to describe MDC indexes. These are mentioned here and on the pages following.

MDC Design

- Choose dimensions with low cardinality
  - More rows per cell
    - Better to have more cells with same values than many partially filled cells
  - Avoid dimension columns on timestamps, account numbers, etc.
  - Consider generated columns
    - If you can afford the overhead during load, etc.
- Can even build MDC using only one dimension
  - Possible alternative to standard clustering index
- Choice of TS extent size is important
  - Larger extent size reduces I/O – match to prefetch size (or automatic)
- To clean up space, **REORG TABLE** with **RECLAIM EXTENTS**

36

Early recommendations point to lower cardinality choices for dimensions, with a large number of rows in each cell. Generated columns will help; for example, use month and year as a dimension rather than days. A timestamp would be a bad choice as each value would be in a different cell.

MDC indexes can also be built using a single dimension. This could be a good alternative to traditional clustering indexes, particularly when wanting to avoid reorgs.

Think about:
Non-Clustered Indexes versus Prefetch

- If prefetch can read 32+ pages in one I/O
  - Many rows read with a large BP and table compression
- How many rows are read in a single prefetch operation?
  - Consider rows per page (or rows/block)
- What is the minimum cardinality for a non-clustered index to be useful?

With DB2 being able to read a large number of rows efficiently, this raises the question shown. How good must your non-clustering index be to provide a better access path choice for DB2?

Minimum Fullkey Calculation

- Consider a 150M row table on 2M pages
  - 75 Rows per page
- Prefetches to read table:
  - Using prefetch size 64: 31,250 prefetch operations
    - 75 * 64 = 4,800 rows read per fetch
- If data is completely unclustered and assuming uniform distribution
  - Index cardinality must be > 4,800 to be better than prefetch
    - Otherwise, just use prefetch → At least one row qualifies in each operation
    - Must also add the index I/O as well
    - Prefer to have cardinality much higher – in this case, 50K or more

This calculation shows how prefetch really helps I/O, and how non-clustered indexes must have a decent cardinality. In this example, you want to avoid having to read every extent, which is to say, your prefetch quantity. Otherwise, Db2 may calculate that it is cheaper to avoid index I/O and simply use prefetch on the table.

Summary

- Table prefetch can cost (much) less than poorly designed indexes
  - Prefetch often is cheaper than poor indexes
- But good indexes are the better solution
  - Fix your indexes
- Add a block-based area to your buffer pool

First and foremost, good indexes that match the requirements of your SQL should be your objective. If your indexes are poor, Db2 will not use them.

We should not only optimize indexes, but also optimize buffer pools to have optimum prefetch.

Speaker Biography

Martin Hubel is an independent consultant and has worked extensively with DB2 since 1985. Martin develops and teaches DB2 advanced courses and is recognized as a leading authority in the field. He has been using Db2 on Linux, Unix, and Windows since 1993 and has participated in several beta test programs for these platforms. He is an IBM Gold Consultant, IBM Champion, and a member of the IDUG Volunteer Hall of Fame. He is also a member of the Db2 LUW SAP Technical Leadership Exchange.

If you are reading this while in Malta or afterwards, Malta was Martin's 32nd country doing Db2 consulting, training, or speaking.