
Performance problems come in many flavors, with many different causes and many different solutions. I've run into a number of these that I have not seen written about or presented elsewhere and I want to share these with you - not just the specific problem/solution, but also the process of identifying and solving the problem.

Joe Geller has been a Db2 consultant since Version 1.2. He's been a Data Architect, DBA, Data Modeler and Software Engineer, often simultaneously. He feels that the best way to learning one aspect of a system is to work on all aspects. Joe specializes in performance tuning. He once rewrote a Peoplesoft view and improved the query by a factor of 70,000. Joe is the author of two books on database systems - DB2 Performance and Development Guide and IMS Administration, Programming and Data Base Design. Currently Joe is working as a database performance specialist for JPMorgan Chase. Joe is an IBM Information Champion and a leader on the IDUG Content Committee.

# Objectives

- Identify the performance concerns with design choices for keys, indexes and partitioning choices. Should your partitioned tables have partitioned indexes or global indexes?
- Discuss the subtle problems with HASH Joins. We'll also look at Deterministic UDFs - if your UDF is deterministic, make sure you tell Db2 about it.
- Selectivity and Filter Factors - how to help the Optimizer get better estimates in places where it cannot know the cardinality.
- Reorgs and Runstats - how to speed them up. What exactly does an online reorg do (and why does it take so long)?
- How to get the most out of the Package Cache. We'll look at some very important columns that are more important than you may expect. Using the package cache to understand the access path, and using the access path to understand the package cache.

# Arrays

- Common in Programming Languages
- Limited Support in Relational Databases
  - XML
  - JSON
  - NOT as a column in a table
- Array Data Type
- Passed as a Parameter to a SQL or Java Routine



Arrays are a very commonly used programming language construct, but have limited support within relational databases. Although an XML document or JSON can have array elements, tables within Db2 cannot have an array as an ordinary column. Db2 however does support array data types, which can be created (CREATE TYPE statement). These can be used in global variables and local variables declared in compound SQL (such as in a native SQL Stored Procedure) and as a parameter to a SQL or Java routine.

# Arrays

- Example:
- Select the rows from a table(s) where cust_id is one of the elements of an array passed in from the client code.
- The only problem is you can't reference the array directly in the predicate.
  - The solution is to convert the array to a temporary table using the UNNEST builtin table function (it returns a table).

## Arrays

```
CREATE TYPE ID_ARRAY AS BIGINT ARRAY[100];
CREATE PROCEDURE JOE.P1 (IN CUST_IDS ID_ARRAY)
 SELECT CUST_ID,COL1, COL2
   FROM JOE.CUSTOMER C
   WHERE CUST_ID IN (SELECT CUSTID FROM UNNEST(CUST_IDS) AS
  LIST(CUSTID)  )
        AND other predicates
```

# Arrays

## But, there is a (performance) problem:

- Unnest returns a table – what is the cardinality (#rows)?
  - At bind time, Db2 has no way of knowing how many entries there will be in the array. There is no syntax for telling Db2 how many to expect.
    - There actually is a CARDINALITY clause which can be used with UDFs, but not with built-in table functions:
      - Select * from TABLE(MON_GET_TABLE('','',-2) CARDINALITY 100);
  - Therefore Db2 assumes (for cost estimating) the array has the maximum size it is created as (up to 1000).
  - the cost of getting 100 customers will be 100 times as much as getting 1 customer.



Unnest returns a table – what is the cardinality (#rows)? At bind time, Db2 has no way of knowing how many entries there will be in the array. There is no syntax for telling Db2 how many to expect. Therefore Db2 assumes (for cost estimating) the array has the maximum size it is created as (up to 1000). In the above example it will assume there are 100 elements

What's wrong with that? Well, the cost of getting 100 customers will be 100 times as much as getting 1 customer. If you usually only pass in 1 or 2 cust_ids, this is way too high. If there are other predicates and indexes on those predicates, another access path may be cheaper than using the cust_id index if there are 100 entries, but much more expensive than if there is 1 entry.

What can we do? Selectivity to the rescue. The Selectivity clause on a predicate lets you override Db2's filter factor for that predicate. What I do for this situation is to add an unnecessary predicate:

CUST_ID IN (SELECT CUSTID) FROM UNNEST(CUST_IDS) AS LIST(CUSTID) WHERE CUSTID IS NOT NULL SELECTIVITY .01 )

The predicate does nothing (any entries in the array will not be NULL), but that the filter factor is .01. Since Db2 is assuming that the result table has 100 rows, a filter factor of .01 means that only 1 row will be selected from the unnested table.

## Arrays

- What can we do?
- Selectivity to the rescue.
  - The Selectivity clause on a predicate lets you override Db2's filter factor for that predicate.
  - What I do for this situation is to add an unnecessary predicate
  - CUST_ID IN (SELECT CUSTID) FROM UNNEST(CUST_IDS) AS LIST(CUSTID) WHERE CUSTID IS NOT NULL SELECTIVITY .01 )
  - filter factor is .01
  - Since Db2 is assuming that the result table has 100 rows, a filter factor of .01 means that only 1 row will be selected from the unnested table

## Selectivity – Other Use Cases

- Sometimes Db2 just cannot get a good estimate for a predicate
  - Host variables or parameter markers
    - Db2 must use an average value
  - Range predicate with a host variable is the worse situation
- Reopt – let Db2 pick the access path at run time – overhead
- Selectivity to tell Db2 what the filter factor is

8

There are times that Db2 just cannot get a good estimate for a predicate, no matter how thorough your runstats was (including distribution stats, column groups, etc.).

Using host variables or parameter markers rather than literals is a good practice for dynamic statement reuse and for flexibility in coding for static SQL. One statement can be used for different values in the predicates. But with host variables, Db2 cannot use distribution statistics to get a more accurate estimate. It must use an average value (based on the cardinality of the column).

A range predicate with a host variable is an even worse situation. Are you asking for a small range or a very large range? The filter factor that Db2 uses depends on the cardinality of the column. A high cardinality column (e.g. Shipped Timestamp) results in a low filter factor. But, often for a timestamp column you want a large range – a day, a week, a month (e.g. find all orders shipped in the last month).

For these situations, you could use REOPT which will have Db2 calculate a new access path each time the statement is executed, but that adds overhead. If you know the predicate will be satisfied by a particular percentage of the rows, then Selectivity is a very simple and effective solution.

# Selectivity

- SELECT * FROM ORDERS
  - WHERE CUST_ID =? AND SHIP_TS > ?
- Indexes on CUST_ID and SHIP_TS
  - These are large customers with hundreds of orders, but there are thousands of orders per day. If we want orders shipped in the last week, it would be better to use the customer index.
  - If the cardinality for SHIP_TS is 1,000,000, the filter factor is 3/10,000 = .0003
  - There are 1,000,000 rows in the table so Db2 is assuming 300 rows will meet the 2nd predicate. But if we are asking for the last week and there are 3 years of data, then the actual filter factor is 1/156 = .0064 (and 6400 rows will meet the predicate

## Selectivity

- Drawbacks
  - Selectivity is a moving target
    - The actual filter factor may change over time
    - If the ff stays the same but the table grows, the #rows returned will increase. This could throw off the balance with other tables in a join
  - It is so easy to add to a query, if the developer is not good at determining what value to use, you can easily make a mistake
- Do you need to be exact?
  - Not always. In fact, sometimes lying to Db2 can be useful.
- Selectivity vs Optguidelines
  - Selectivity gives Db2 more information
  - Optguidelines **tell** Db2 what access path to use



What do I mean by lying? If Db2 calculates a FF of .03 and you know it is closer to .2, you could use Selectivity .2. But, Db2 may still pick a different access path because other FF estimates are also inexact. With experimentation you may find that using Selectivity .9 will lead to a better access path. It should be possible to achieve the same result by using Selectivity on more of the predicates, but it can be complex. The simpler solution of adjusting 1 predicate has appeal. Of course you should put in a comment describing what you are doing (if you are going to lie, you should at least be truthful about it).

Because of the complex interplay between multiple predicates and filter factors, it is sometimes hard to get the exact access path you want (which doesn't mean what you want is really best). Optguidelines allow you to tell Db2 what access path to use. But of course, you are then taking Db2's intelligence out of the equation. If you are not smarter than Db2, you can ask for a worse performing access path.

# Optguidelines

- SELECT COLA FROM TAB1 T1, TAB2 T2
  WHERE T1.COLB = T2.COLB
  /*<OPTGUIDELINES>
  <HSJOIN>
  <TBSCAN TABLE='T1'/>
  <TBSCAN TABLE='T2'/>
  </HSJOIN>
  </OPTGUIDELINES> */
  ;



The guideline is in XML format and in the form of a comment /*….. */
It must be the last part of the statement.

# Optguidelines

- SELECT COLA FROM TAB1 T1, TAB2 T2
  WHERE T1.COLB = T2.COLB; **XXX – this will not work**
  /*<OPTGUIDELINES>
  <HSJOIN>
  <TBSCAN TABLE='T1'/>
  <TBSCAN TABLE='T2'/>
  </HSJOIN>
  </OPTGUIDELINES> */



If the optguideline is after the ; then it is not part of the statement and is ignored.

## Optguidelines

- SELECT COLA FROM TAB1 T1, TAB2 T2
  WHERE T1.COLB = T2.COLB
  /*<OPTGUIDELINES>
  <HSJOIN>
  <TBSCAN TABLE='T1'/>
  <TBSCAN TABLE='T2'/>
  </HSJOIN>
  </OPTGUIDELINES> */
  WITH UR; **XXX – this will not work**



If the optguideline is not the very last part of the statement, it is treated as a comment and is ignored.

# HASH JOINS

- Good for batch processing – reading of whole tables
  - A nested loop join will be good if the tables are both clustered on the join columns. Otherwise you get a lot of random I/O
  - A hash join reads the tables with sequential prefetch. You get asynchronous I/O, typically with 32 pages per I/O
    - 400,000,000 rows read with prefetch can be faster than 1 million randomly read rows

# HASH JOINS

- A hash table is built from the selected rows of the inner table
- The size of the hash table (the # of buckets) is based on the estimated # rows
  - This is dependent on the statistics and the filter factors
  - It is essential that this size be accurate.
    - If too small, you get very high CPU utilization as there will be long chains of rows that hash to the same bucket
  - Generally it is also better for the smaller (key + selected columns) table (after filtering) to be the inner table (right hand side of the access path).
    - The hash table comes from SORTHEAP. Make sure your sortheap is big enough



When multiple rows of the inner table hash to the same bucket, a chain is built of all of these keys. When a row of the outer tables hashes to that bucket, Db2 must search that chain for the matching key.

## Beware of : Grand Pronouncements and Generic Solutions
## Beware of : One Size Fits All

- One Big BP is better than many smaller ones
  - So you should go with one big bp
  - The first part of this statement has some truth to it, the second does not



IBM (and other experts) sometimes give tuning advice as though one solution or best practice, will fit all customers. But each db is unique – some are designed well (tables, indexes and keys) and some are not. Access patterns also vary greatly even within one db.

# Bufferpools

- There are well established cases for some separation of data into separate bufferpools
  - Separate the catalog
  - Very different access patterns
    - Separate tables and indexes
    - Temporary tables
    - Very large tables (perhaps bigger than your total bp size) that are mostly processed sequentially

# Bufferpools

- There are well established cases for some separation of data into separate bufferpools
  - Very different access patterns
    - Very large tables (perhaps bigger than your total bp size) that are mostly processed sequentially
      - Separate small bp – since each scan will more than fill the bp, you will not get any reuse for the next scan. Small so you don't waste memory
    - Very large table with many tb scans, but also random access
      - Use a shared bp, but define a block based area for the prefetched blocks of data
      - ("block based area are of no actual benefit" is another generic pronouncement)

        However, see **DB2 LUW 'PERFORMANCE FIRST AID' WITH MONREPORT.DBSUMMARY**

        http://www.idug.org/p/bl/ar/blogaid=625



Phil Nelson and Martin Hubel have had excellent results from using a small block based area (sometimes as small as 1000 pages). IBM doesn't seem to get this and has talked about removing this feature. However, in his most recent article for the IDUG Content Blog, IBM's Steve Rees did conceed that it could sometimes be useful (**DB2 LUW 'PERFORMANCE FIRST AID' WITH MONREPORT.DBSUMMARY**
http://www.idug.org/p/bl/ar/blogaid=625).

# Bufferpools

- What is behind “use one big bp”?
  - Other than separating tables and indexes and the catalog, other separation requires detailed analysis of access patterns and bp performance, along with extensive testing and monitoring
  - Without that effort and expertise, one big pool for data and one for indexes is better. If you split into smaller pools (perhaps by application or schema) and those pools have different peak times of day, that memory is wasted the other times. One bigger pool will make better use of memory and service the data more efficiently

# Partitioned Indexes

- Range Partitioned tables
  - A number of benefits – especially if you choose the partitioning column(s) well.
    - Easy purging / archiving of old data
    - Possibly reduce amount of data to search to satisfy a query
- Indexes on a partitioned table can be partitioned or non-partitioned (global)

## Partitioned Indexes

- Grand Pronouncement – all your indexes on partitioned tables should be partitioned
- Wrong – not always
- Partitioning column(s) are often chosen to make purging easy
  - Partition rollout – detach
    - Very fast if all of the indexes are partitioned
- Separate index trees for each index partition – index tree may be smaller (fewer levels)

# Partitioned Indexes

- Drawback – more costly online queries
  - If all the queries have a predicate on the partitioning column, then you get partition elimination and only those partitions which meet that predicate are searched
  - If a query does not, then all partitions are searched (or if the predicate is too big a range, then many partitions are searched).
  - That means that if you have 20 partitions, there will be 20 index tree searches instead of 1. For online transactions, that is not good.
  - So, choose the partitioning columns carefully



For example, for purging purposes you might have chosen last_update_date or created_date as the partitioning key. But if the queries typically use ship_date as a predicate, you do not get any partition elimination. It would be better for query performance to be partitioned by ship_date.

Of course, some of the queries may not have a predicate on ship_date either. If that is the case, you should see if such a predicate could be added. For example, if you are inquiring on orders ordered in the last week or month, then ship_date must also be greater than that (or NULL). So a predicate on ship_date could easily be added.

# Index Design

- Business vs Surrogate Keys
  - The biggest factor in performance is I/O
  - Joins perform best if the 2 tables are both clustered by the join column
  - Ex. Join of Customer to Order by cust_id.
    - If the Order table is clustered by order_id, the orders for a customer will likely be stored on different pages.
      - If you then join to Order_item table and that table is clustered by item_id, each of those rows will be on different pages
    - The best design for performance would be to have cust_id in all 3 tables and cluster each table by cust_id

# Index Design

- Non-unique Indexes
  - There is one entry per key value with a list of RIDs for each row. These rids are maintained in rid order.
- Low cardinality Indexes have some major drawbacks, especially if the key is frequently updated (such as STATUS).
  - Long chain of rids – to update the key, Db2 must search the chain to find the rid for the row being updated. It then has to delete that rid and insert a new entry (in the right place in the chain) for the new key value
  - Adding additional columns (such as id columns) to the index will reduce the size of the rid chains, and also provide more predicate filtering in the index

# Index Design

- Additional columns in low cardinality Indexes (cont)
  - Will likely also reduce lock waits because entries are more spread through the index
  - Why do some people have such low cardinality indexes?
    - To find the “open” or “pending” rows
    - Indexes are created on a variety of columns that have predicates
      - Do not fall into this trap – if the column is not the primary search criteria, this index is most likely useless



Why do some people have such low cardinality indexes? Typically it would be to find the “open” or “pending” rows for batch processing. In this case, it is necessary to have distribution stats on the column and to use literals for the predicate. Otherwise Db2 has to assume a uniform distribution.

# Index Design

- Index on Expression
  - CREATE INDEX UNAME ON CUSTOMER (UPPER(LASTNAME))
  - SELECT * FROM CUSTOMER WHERE UPPER(LASTNAME)='BAKER'
  - !!!But – you cannot do an Inplace Reorg on a table with an IOE!!!
  - I have opened an RFE (Request for Enhancement) to allow an inplace reorg on an Expression Based Index:
    - https://www.ibm.com/developerworks/rfe/execute?use_case=viewRfe&CR_ID=114749



Index on Expression was introduced in V10.5. Without IOE (also called functional indexes), in order to get index access you had to add an additional column (derived on the expression) to the table and create an index on that column.

# Index Design

- Index Include Columns
  - Adding additional columns to an index can enable more filtering before reading the data row, and possibly index only access
  - A unique index on order_num, ship_date will not be able to enforce uniqueness on order_num
  - An include column is a column in a unique index that is not part of the key
    - CREATE UNIQUE INDEX ORDERKEY ON ORDER (ORDER_NUM) INCLUDE (SHIP_DATE)

## Indexable Predicates

- Some of the following query rewrites have been available since version 9.7, others have been added since then.
  - YEAR(date_col) = 2016 – becomes date_col between '2016-01-01' and '2016-12-31'
  - DATE(ts_col) = '2017-08-13' - becomes ts_col between '2017-08-13-00.00.00.000000' and '2017-08-13-23.59.59.999999'
  - CURRENT_DATE BETWEEN col1 and col2 - becomes col1 <= CURRENT_DATE and col2 >= CURRENT_DATE
  - SUBSTR(col,1,3) = 'SMI' - becomes col1 >= 'SMIxxxxx' AND col1 <= 'SMIyyyyy' where the x's represent low values and the y's represent high values.

28

In general, using a function or expression on a column in a predicate makes that expression non-indexable. This was always true up until a few releases ago. If you have an index on that column, you could manually write the predicate to avoid the use of the function or to move the expression to the other side of the operator. Now, during the query rewrite phase, Db2 can rewrite some of these predicates into a form that is indexable.

# Current Date Filter Factor

- Difficulty with recent Date / Timestamp filter factors
  - The less recent the Runstats, the less accurate the FF
- Current Date is treated differently for Static vs Dynamic SQL
  - Dynamic – the actual date/TS is used for the FF calculation
  - Static – treated as a host variable
  - Neither is completely accurate



This is a major limitation of any dbms – to accurately estimate the FF for predicates comparing a date / timestamp to the current date / timestamp.

# User Defined Functions – Deterministic / Non-Deterministic

- A deterministic function is one that always returns the same result for identical input parameters.
- Deterministic function -
  - Function IS_Unprocessed_Order (P_STATUS INT)
  - IF P_STATUS IN ('OPEN','PENDING')
    - THEN RETURN 1
    - ELSE RETURN 0
- Non-Deterministic function –
  - Function AGE (P_BIRTH_DATE DATE)

# User Defined Functions – Deterministic / Non-Deterministic

- If Db2 knows the UDF is deterministic, it caches inputs/results and avoids invoking the UDF if the input is cached. This is a large CPU savings
  - The default is non-deterministic
  - The keyword DETERMINISTIC on the CREATE FUNCTION tells Db2 that the function is deterministic
  - This keyword can only be used on a standalone UDF, not one inside a Module

## Inplace Reorgs – Speeding them up

- Inplace reorgs allow concurrent read/write activity to the table
  - But they can be slow
  - Re-establish clustering (if there is a clustering index)
  - Eliminate overflows
  - Re-establish freespace (if PCTFREE > 0)
  - Fill in extra freespace

32

If you have a clustering index, Db2 will attempt to insert new rows near the existing rows in key order. If there is no freespace on that page, the new row will have to go on a different page. A reorg will move the data rows around to store them in strict index order. Setting PCTFREE > 0 for the table will leave freespace on each page during the reorg (so that new index inserts can find room on that page).

Over time, any freespace left during a reorg (the % specified by PCTFREE) will get used by newly inserted rows and updates to existing rows that increase the size of the row.

The reorg will not only put the rows in clustering order (if there is a clustering index), but it will also move rows from a page that no longer has PCTFREE percent of the page free (because it has to bring each page back to that percent of freespace).

If there are pages with more than PCTFREE freespace (because of row deletes), Db2 will move rows to these pages to reduce the amount of freespace back down to PCTFREE.

Overflows are created when a row is updated, the new length of the row is larger, and there is not enough room on the page for the enlargened row. The row is moved to another page, but an anchor is left on the original page (pointing to the row's new location). This way, the indexes do not have to be updated to point to the new location. A reorg will eliminate overflows by updating the indexes and removing the anchor.

## Inplace Reorgs – Speeding them up

- The ones we really care about are clustering and overflows
- If you mostly care about overflows, then:
  - CLEANUP OVERFLOWS (new with Db2 10.5)
  - If 10.1 or earlier, use PCTFREE 0 – no movement of rows to make freespace, remove excess freespace only if many deletes have been issued, so limited movement of rows to fill in freespace

33

Some shops do not have clustering indexes (or may have them on some tables but not all). Even if you have a clustering index, the table may not get out of clustering order too often for several reasons and overflows are a bigger concern. The rate of inserts might be low, but updates occur frequently. Or, online access to them reads only 1 or 2 rows so it is not critical that the rows be in clustering order.

Overflows on the other hand are a concern, especially for batch processing (table scans). Each read of a row that has an overflow requires access to 2 pages – the original (which has the anchor) and the one with the actual row. For a single row that will mean 2 logical reads instead of 1 and possibly an extra physical read (if the page is not in the bufferpool). 1 extra read does not take a lot of time. A table scan on the other hand reads all of the pages with sequential prefetch, which reads multiple pages asynchronously. When it finds an overflow, it has to do a synchronous read of the new page (if not in the bufferpool). The relative effect is thus much greater.

# TABLESAMPLE

- Runstats option
  - Samples a percentage of the rows (TABLESAMPLE BERNOULLI) or the pages (TABLESAMPLE SYSTEM)
  - Greatly speeds up the runstats
    - SYSTEM is faster because it does less I/O, but less accurate
    - The lower the percentage, the less accurate the statistics

# TABLESAMPLE

- Can be used on SELECT statement too!
  - An approximate count of a large table
  - A set of test data, but you don't want it all from the "beginning" of the table because over time the data values have changed

# PACKAGE CACHE

- MON_GET_PKG_CACHE_STMT –
  - Table function to read the package cache
- Best (built-in) tool for analyzing the performance of individual statements
  - 1 row per stmt (static or dynamic)
  - Package Cache has two use cases:
    - Primary use - dynamic stmt reuse – avoid prepare
    - Secondary – metrics

# PACKAGE CACHE

- Best (built-in) tool for analyzing the performance of individual statements
  - Cumulative
    - Since Db2 recycle
    - if full, Statements are flushed to make room for others
    - If the statement cannot be reused (rebind, runstats, has a global temp table, etc.), then flushed
    - I have opened an RFE to retain statements in the cache even if they cannot be reused:
      - https://www.ibm.com/developerworks/rfe/execute?use_case=viewRfe&CR_ID=114027

37

If you want to get every single statement and not miss the ones that are flushed, you can use a package cache event monitor to capture all statements.

# PACKAGE CACHE

- Dynamic SQL should use host variables or parameter markers for ID columns (or other high cardinality columns). Otherwise the cache is filled with statements that won't be reused - forcing out others.
- Many columns – metrics for num execs, CPU, elapsed, wait times – lock wait time and counts, I/O - broken down into data and index and temp; logical and physical reads



Every unique dynamic statement get a row. If there are variation such as different status codes or type columns, you will get a few rows in the cache. But if literals are used for ID columns, then every single execution will be a unique statement and will get a separate row in the package cache.

# PACKAGE CACHE

- Perf Monitors? DSM, DBI, others
  - Some, like DSM (which is not fully mature) give you the columns that the *author thought* were important
  - I want all columns available
  - Why?
    - Many causes of poorly performing queries
      - Many symptoms
    - Many solutions
    - Some are straight forward; some are not
      - Any given problem may require looking at metrics that the tools don't give you

# PACKAGE CACHE

- MON_GET_PKG_CACHE_STMT is a table function, so you issue SQL against it; with a where clause to pull out:
  - Single stmt based on stmt text
  - All stmts for a package (SP)
  - Stmts meeting other criteria – total_cpu_time > ?
    - avg - total_cpu_time / num_execs > ?
  - Order by cpu desc, elapsed_time desc, etc.
- Select the entire package cache and export it to a csv file to open in Excel
  Can sort the columns, find stmt_text, package_name, etc.
  - If you do the Select in Data Studio, you can also easily sort by any column

# PACKAGE CACHE

- Which columns should you Select?
  - SELECT * - easy, but with each release the # of columns has grown
    - V9.7 – 64 columns
    - V11.1 – 277 columns
    - 277 is very cumbersome to work with
      - A large file
      - A lot of scrolling left and right
  - SELECT column list (of the ones you want)
    - Easier to work with
    - Many are not applicable to your database
      - BLU
      - pureScale
    - You may miss some that you need
    - More work to setup and maintain

# PACKAGE CACHE

- Which columns are most useful?
  - Two Use Cases
    - Poorly performing stmt – deep dive
    - Identify poorly performing stmts – find candidates
  - Different columns are of more use for one or the other or both use cases

# PACKAGE CACHE

- Performance is based on time spent
- There are metrics for that
- Other metrics are useful for analysis, but do not represent actual time spent and therefore do not directly indicate performance

# PACKAGE CACHE

- When looking for poorly performing stmts I look for time spent metrics, but not the others
- When analyzing a statement which has been identified as poorly performing, then I look at the others

# PACKAGE CACHE

- Rows Read / Rows Returned
  - Sometimes called Read Efficiency
  - A high ratio does **not** always mean a problem
    - A batch job reading the whole table, will often just have predicates on non-indexed columns. There will be a high number of rows read
    - Select count(*) will read the whole table, but only return 1 row

# PACKAGE CACHE

- Rows Read
  - Who cares how many – if done efficiently; or if they are needed. The program with the most number of rows read maybe needs to process all of those rows
  - It also depends on the access path
    - A hash join (reading both tables completely) will often out perform a nested loop join, depending on the clustering of the two tables
      - Asynchronous I/O (prefetch) vs random synchronous I/O
      - The hash join will have many more rows read. Reducing rows read is not the goal, reducing elapsed time is
  - But, it could indicate a problem for a given statement

# PACKAGE CACHE

- Logical Reads
- Similar to rows read – a tbscan will do a logical read of all the pages
  - This may be okay. But, extremely high rows read or logical reads could indicate a bad access path
  - you shouldn't look at clues in isolation – they fit together
  - Look at your Explain along with the package cache
    - If a query has 3 large tables totaling 500 million rows, but the package cache shows 2 trillion rows read, this might indicate a nested loop join in which the access to the inner table is not using an index efficiently
    - That is a problem

# PACKAGE CACHE

- Hit Ratio
  - a ratio is not a measure of time spent
  - An I/O (physical read) does take time
  - Elapsed time is affected by the number of physical reads
    - 10,000 logical reads and 2000 physical reads – 80% hit ratio – not too good
    - 1,000,000 logical reads and 50,000 physical read – 95% hit ratio – very good;
      - But, 25 times as much elapsed time
  - Hit ratio is useful as a general indicator of bp performance
  - It might mean the bp could benefit from more memory

48

Hit ratio is not a column in the package cache, but it is easy to calculate.

# PACKAGE CACHE

- Physical reads - A high amount might mean
  - Small bufferpool
  - Poor table or index design (clustering, surrogate generated keys vs business keys)
  - You can't design optimally for all use cases: tables can be clustered by only 1 index (except MDC tables)
  - Program design
    - Batch program – if the input is not ordered by the index order or the clustering order of the tables, it will get many more logical reads and probably more physical reads. Db2 will have to jump around the table doing more random access
    - Reading more data than necessary

# PACKAGE CACHE

- Elapsed Time
  - There are two metrics for Elapsed Time
    - STMT_EXEC_TIME
      - Time spent in this statement plus everything invoked by this statement
      - (SPs and UDFs invoked)
      - Useful for identifying long running Calls
    - TOTAL_ACT_TIME
      - Time spent in this statement
      - Useful for identifying which statements within the SP take a long time

# Getting More out of the Package Cache

- Queries
  - Predicates – date, package_name, stmt_text, num_exec, total_act_time, total_cpu_time
  - Order By
  - Data Studio – can re-sort by any column
  - Export to a CSV file and open in Excel. You can sort, find (stmt_text, package_name)

# Getting More out of the Package Cache

- Cumulative – not individual executions; totals depend on num_exec
  - So, get averages. Divide your favorite columns by num_exec

# Getting More out of the Package Cache

- Point in Time – You need deltas to get a narrower time period
  - Collect twice – or daily or 1/hour 1/5min 1/min
    - Look at 2 consecutive rows and calculate the delta. How?
      - Manually
        - Excel – subtract
      - Load into a table along with the TS of when selected
        - Query joining to itself to get delta

# Getting More out of the Package Cache

- Point in Time – You need deltas
  - Better
    - Calculate delta when inserting into table
      - Now can get delta easily with a simple select
      - Can be deltas of total or avg of deltas
    - Trends / spikes
      - Query joining 2 rows where 2nd is X% more than first

IDUG
Leading the DB2 User Community since 1988

IDUG DB2 EMEA Tech Conference
Lisbon Portugal | October 2017

#IDUGDB2

# Some of My Favorite Tuning Tips and Tricks for Db2

**Joe Geller**
*JPMorgan Chase*
JoeDB2@aol.com

Session code: C16

*Please fill out your session evaluation before leaving!*