

## Designing and Implementing VLTBs for Performance

By Susan Lawson, YL&A

What is considered large today is certainly not the same as what was considered large a few years ago. What is a large table today? Maybe somewhere at billion rows, but as we have experience in the last few years is that multi-billion row tables are becoming more popular. But regardless of whether a table is 2 billion rows or 45 billion rows there are some new thing to thinks about. Our design criteria changes with tables of this size. They have to be designed differently if they are going to be maintained. We have to be very careful when analyzing all types of processing that will occur against the table because this will have even more of an impact on our design. Sequential/range retrieval and SQL access become a secondary issue whereas maintenance, recovery, and use become the primary focus.

Partition strategies need to be carefully thought out and this is NOT something you will see ever come out of a data modeling tool or out of the logical design process. You many want to consider usage an ascending key for data over time, or possibly the use the V6 ROWID for random linear distribution (we look at each strategy later in this paper).

In addition to all of the advantages provided by partitioning table spaces, for large tables partitioning is the only way to store these incredibly large amounts of data. But do not lose site of the fact that partitioning also has advantages for tables that are not necessarily large – but require parallel access and better availability. DB2 allows us to define up 254 partitions up to 64 GB each. Non-partitioned table spaces are limited to 64 GB of data. Partitioning will allow you to take advantage of query, CPU, and sysplex parallelism. Even defining a table with one partition will allow a query involving a join to enable CPU parallelism. You can take advantage of the ability to execute utilities on separate partitions in parallel and in Version 7 we will have the ability to LOAD multiple partitions in parallel one job. Partitioning also gives you the ability to access data in certain partitions while utilities are executing on others. In a data-sharing environment, you can spread partitions among several members to split workloads. You can also spread your data over multiple volumes and need not use the same storage group for each dataset belonging to the table space. This also allows you to place frequently accessed partitions on faster devices.

However, even with all the advantages of having partitioned table spaces, you must keep a few potential disadvantages in mind when deciding about partitions. For instance, unless you are at the version 6 level, you cannot use the ALTER statement to change the partitioning key ranges. More importantly is the fact that we still cannot add additional partitions to a table (however, this is a very high priority item for DB2 development and hopefully we will see it in the future). This could be a problem if your tablespace has become large and has run out of space, it would force you to have to drop, recreate, and reload. With the large multi-billion row table this is simply not an option.

### *Some Partitioning Strategies*

There are several strategies for partitioning your data. You will need to choose a partitioning strategy based on the nature and size of the data being stored and its intended use. We will take a look at some of the more popular partitioning strategies.

### ***Linear Distribution***

You can design partitions to allow for linear distribution of data. This can be difficult with mismatched size partitions and partitions can run out of space. In order to be able to do this it will require a truly random key and be a well thought out design. One option is to use the ROWID column data type (introduced in V6 for support of LOBs.). The value of ROWID is a random value in the range from low-values to high-values (hex '0000...' through hex 'FFFF...'). Therefore if using a ROWID as the high level column of a partitioning key, you can have linear distribution over the partitions in a partitioned tablespace. This can also be useful to take advantage of direct row access – therefore bypassing several levels of an index (which can be 5 and above with some large tables).

However, there is one very serious disadvantage to random distribution due to the key normally used for linear distribution. Most often this random key is some type of surrogate key, meaning that it has no natural key value for the row. While this might work in OLTP environments, it normally gives rise to the “death by random I/O” for any kind of work involving sets of data from the table. In reality, OLTP can suffer dramatically when sets of data are required for two common scenarios. One is when result sets are retrieved for scrolling, a technique applicable in almost every application. The other is where sets of data are required for any transaction. One very common design strategy that mandates this kind of access is where updates are not applied but stored as a new record with a forward ascending timestamp. Another situation is where the most current row needs to be retrieved from a set of like rows. There is only one row returned but many rows will need to be examined in order to return that one row. As tables get larger and larger, the linear distribution is potentially the worst method for adding additional data.

Also often when developing large tables a surrogate key is developed to decrease the size of a key (when a natural key is comprised of several columns). This is not an optimal design and you could be sacrificing performance for DASD savings. DASD is always getting cheaper and should not become an inhibitor when implementing a large table, especially if it means sacrificing performance.

### ***Ascending Sequence***

DB2 handles this very well and there are generally no problems with hot data areas. If data is always added to the end this strategy can remove maintenance problems with older partitions and this is very useful when dealing with large tables.

While this works extremely well for inserts, there can be a problem that requires some well thought out design of this technique and that is where there are multiple update processes and delete processes referencing these new rows. One example where this occurs is where the data that has been inserted, needs to be validated, edited, and updated before being used. In addition, when staging tables are used in this way we will compound the difficulty by adding deletes to the mix. This is one case that row level locking might be required – where there are many multiple insert, update, and delete processes occurring on or near the same area. There are some extreme techniques that can be used for data spreading to eliminate this but they can have a negative effect on sequential processes. This is another case where the primary use needs to be the driving factor of the physical design and not the logical design key construct.

### ***Archiving/Rolling***

If you are using each partition to hold a periods worth of data, for example a month per partition, and later archive the older partitions as a new period begins, then are several maintenance issues to deal with. This was a popular design when DB2 could not hold large amounts of data, but that is becoming less of a issue with each release. We are often able to hold more historical data in the same table without having to roll off older partitions. Archive strategies will probably never go away, but maybe we can lessen the frequency.

### ***One partition table strategies***

A one-partition table was necessary in V5 in order to have the table qualify for query parallelism since there was no parallelism for segmented or simple table spaces. This is very useful when a table will be used in any joins because DB2 will split the query into multiple queries and run them parallel. In Version 6 and beyond this will not be necessary since all tables can participate in parallelism.

### ***Selecting a Partitioning Key***

The selection of a partitioning key will be based upon the strategy you have chosen for implementation of your table space. One item to note is that as of Version 6 you can update partitioning keys through a SQL update rather than by performing an INSERT and DELETE. This is not something that should be used often and you should aim to use a partitioning key strategy that does not need to be updated. The reason you do not want to be updating partitioning keys often, is because of the fact that DB2 will have to determine what partition the new value will need to go in, so it will drain the range of partitions effected and the entire NPI(s) on the table space. This could cause major performance problems in a high volume OLTP environment. This impact may be lessened if the column being updated is not one of the columns controlling the partitioning strategy that is the column is not one of the leading columns that will be compared to the key range.

So the guideline here is to use the SQL INSERT/DELETE strategy in a single UOW when you can for high performance. The SQL UPDATE is a nice feature for code generated systems, and other systems that might be dealing with data that could or could not be partitioned. There are many times when a system was developed, that normal SQL UPDATE statements were used. But over time as these systems grow in size they might have to be converted to partitioned table spaces. With this V6 feature, the code does not have to be changed. This also increases the portability of the application. But here we are dealing with performance, and if you want high performance and high availability then use the INSERT/DELETE strategy, especially if you have NPIs on the table space.

### ***Re-Balancing Partitioning Keys***

Prior to Version 6 there was a problem sometimes referred to as ‘ballooning partitions.’ This would occur when the data simply outgrew the key range defined to the partition and there was no ‘easy’ way to enlarge the partition or to redistribute the key values. With an enhancement that arrived in version 6, we can redefine the range values of partitions. Then through a

subsequent reorganization of the table space, we can re-distribute the data. This ability gives rise to some new design strategies. Quite often just simply having the ability to rebalance is not enough. We still cannot add partitions, but if we were to create a table space with ‘free’ partitions then if individual partition growth became a problem it could be handled by redistributing the data into ‘free’ partitions, thus accommodating the expansion.

However, with the introduction of partitioning key rebalancing comes new issues. This will cause an outage because the partitions will be placed in a REORG pending status and the range will have to be reorganized before the table space can be fully used by the applications. Depending on the altering of the key ranges, this may affect several partitions and associated non-partitioning indexes.

## **Non-Partitioning Indexes**

For large tables NPIs DO WORK! The trick is designing them properly so that they do not become a maintenance nightmare.

Partitioning indexes are sometimes used for all data access of a partitioned table and other times are solely used for a partitioning sequence to enable ease of use, growth, and maintenance. It is very rare that a single index will suffice for both random and sequential access to a partitioned tablespace in support of both OLTP and batch access. In all these cases, there is often a strong need for NPIs and yet as tables get larger and larger, NPIs are feared and in many cases not allowed. Remember, there is never an always rule but in many places we have been we have found that rule in place. It normally comes from a bad experience with NPIs. That also is not unusual since it is easy to have a bad experience from NPIs since they do need extensive planning if they are to be effective.

NPIs can be good or bad depending on whether or not the keys are ever updated. Another dependency is whether or not PIECESIZE is used and the definition of the pieces, which will be discussed below. The maintenance of NPIs and the underlying tablespaces is quite different since a partition and a partition index is the granularity for utilities, but an NPI’s granularity is the entire index. The larger the tablespace is, then the more difficult it is dealing with utilities and partition independence, recovery, and online reorgs. Many of the problems with NPIs can be alleviated by designing the underlying tablespace and the partitioning key in such a way that the NPI would be built using key columns that are permanent – non-updateable. There are enhancements coming in the future for NPIs (such as improvements to the Online REORG BUILD2 phase in Version 7) to make them less difficult to deal with, but until that time it just takes planning in order to have use outweigh the difficulties.

### ***PIECES***

The PIECESIZE option on an index creation allows you to break a non-partitioning index into several datasets (or “pieces”). Using pieces allows for better performance of INSERT, UPDATE and DELETE processes by eliminating the bottleneck previously caused by only having one dataset behind the index. The use of pieces is essential for non-partitioning indexes that support large tables, but it can also be helpful for concurrency performance of heavy INSERT, UPDATE, DELETE processing against any size partitioned table space with NPIs

An individual piece can be defined to be as large as 64GB and we can have up to 254 (making a 12,256 GB NPI !); however, for practical purposes the PIECESIZE will normally

be set so that a piece will fit on one disk volume. This allows more appropriately placed datasets on different disk volumes to relieve contention when the index is accessed. Currently, the use of the piece is limited just to physical partitioning and parallel access.

When a load is performed and the non-partitioning indexes are using PIECESIZE, the entire piece will be filled up to the amount specified in the piecesize definition during the creation of the NPI. DB2 will use the primary space quantity and as many secondary extents as necessary to fill up to the piecesize. During later processing when an extent is necessary, the extent will be taken at the end of the non-partitioning index.

We need to take advantage of pieces for best performance and manageability of NPIs – for large as well as non-large partitioned tablespaces

### ***Sizing Pieces***

To set the primary quantity space definition for a piece of an NPI, first you need to set the size for a piece of an NPI. To do this you need to determine the number of pieces and set the PIECESIZE value. To choose a PIECESIZE value, divide the size of the non-partitioning index by the number of data sets that you want, and this is different for every NPI in every application and would be based on I/O configuration and the sizes of the disk devices and the size of the NPI.

Once the number of pieces and the PIECESIZE is determined, you can define the proper primary and secondary space quantities. The primary and secondary quantities should be evenly divisible in PIECESIZE or else there will be wasted space in the NPI.

Based on user experiences it would be best to define the primary quantity equal to the PIECESIZE and keeping the secondary quantity relatively small since it should never be used.

$$\boxed{\text{PRIQTY} + (\text{max-extents} \times \text{SECQTY}) \Rightarrow \text{PIECESIZE}}$$

Also evenly divisible for full space use divide size of NPI by number of data sets desired. There is the 32 datasets max for non-LARGE indexes.

## **Case Study - Implementation of a 6 Billion Row Table**

The best way to talk about how to implement VLTBs is to discuss an actual implementation and all the considerations and issues surrounding it.

We got the opportunity to design and implement one of the first VLTBs – a six billion row table. Now as we begin to develop tables in the 20-40 billion row range, that number starts to seem low, but the issues that we came across and some of the techniques we used are still very applicable when implementing tables of this size and beyond.

### ***Prior to Version 5***

Large tablespaces have a significant impact on the database world. In the past, a technique commonly used to implement tables larger than 64 GB was to allow several physical tables to implement a single logical table. While this technique worked, it was cumbersome to implement, because it required some kind of finder table in front of the process, or direct processing to the appropriate physical tablespace. In addition, there could not be a real index, as each of the physical tables acting as logical indexes had their own partitioning indexes, and there

was no method for defining a non-partitioned index (NPI). The requirements for something like a non-partitioned index were to have additional tables serve as logical indexes into the multiple physical tablespaces, but each of the "index" tables also required their own indexes. All in all, this got quite messy, as the code to navigate through these layers had to be finely tuned, and most coding had to be encapsulated in application functional modules.

One also had to make extensive use of SET PACKAGE in order to move through the layers; this required having different collections point to different databases, and, in some cases, attaching synonyms or aliases to physical objects in the other databases. If this sounds confusing, you should see the diagrams and object definitions! With the arrival of DB2 Version 5, these problems can be alleviated, as long as you have time to move through the unloads, drops, redefines, and reloads. But it is worthwhile, especially for those systems that are hitting either the 64 GB limit or the partition size limits of 1, 2 or 4 GB since both those constraints are gone with the 1TB tablespaces as of Version 5.

To define a large table of just 100 GB using the old method required a non-partitioning index. We had to define: two partitioned tablespaces, two partitioning indexes, an "index" table for the NPI, a real index on the NPI, and a finder table in front with its own index - a total of seven objects!

As of Version 5, we would only need three objects: the 1TB tablespace, its partitioning index, and the NPI. But if we apply the same example to the implementation of a core table was initially loaded with 5.3 billion rows, the true value of the large tablespace becomes readily apparent. This table has well over 100 partitions, more than 100 partitioning index partitions, and two non-partitioning indexes of significant size. Also, prior to Version 5, the non-partitioning indexes were a real problem, since regardless of the number of partitions, the NPI was a single object; this made it "performance challenged"! One of the NPIs for 6 billion row table was close to 200 GBs.

### ***Designing of the Table***

Probably the most interesting part of this system was not only the fact that it had a core table of 6 billion rows, but the fact that there would be heavy usage of OLTP queries, DSS queries, OLCP(on-line complex) queries, and concurrent batch and online programs. Well, I heard that this could not be done? Sounds like a challenge to me!

The core table had to be defined as a LARGE tablespace of 254 partitions (Version 5 was just GA at this time). Each partition would hold approximately 40,000,000 rows each and the key would be based on date ranges to control the sequencing. The dates of the data to be loaded went back to December 31, 1958 and we picked date ranges for the partitions to be filled in the future up to June 30, 2018. Data would be added off of the upper end so only a few partitions would be active with data being inserted. The other partitions would not be 'active' in terms of new insertions of data. But this is not the way the data was accessed?? What about the queries? At this point, we did not care. The table had to be design with a partitioning scheme that allowed for proper care and maintenance of a table of this size. If the table had been designed solely for proper access it would have been a maintenance nightmare and would never have worked.

There was a thought given to carry a partition id in the table to help allow dynamic control without ballooning partitions, but when you consider that disk space required to carry this ID it simply was not a wise option.

The initial load of the data would occupy the first 134 partitions, even though all 254 were defined to allow for the growth. The last few partitions were the only ones with active

insertion of data and as time progressed they would become inactive as new partitions begin to fill (partitions were based on data sequence and held approximately a months worth of data). The number of rows per partitions was approximately 40 million each.

The access to the table would be through two non-partitioning indexes.

### ***Clustering Index***

The clustering index was solely designed for partitioning – not access. In other words, the partitioning scheme was designed so that all new data was inserted off the upper end and only a few partitions would be ‘active’ with new data and therefore require the most attention. The other ‘not-so-active’ partitions would only require infrequent copies (if updates were made) but would not need to be reorganized.

Due to the fact that no access was done via the clustering/partitioning index, all access would have using the non-partitioning indexes.

### ***Non-Partitioning Indexes***

For this table we implemented two NPIs. The NPIs were used for ALL access to the table. The clustering index was not used to access and a tablespace scan would not work – it took seven days !!

We made extensive use of pieces with the NPIs. When the table was loaded it filled up 78 pieces for the large NPI and 12 for the smaller one. In this implementation the pieces were sized to match one per physical volume; using user-defined methods, flexibility in placement is assured.

While building this 1TB table, we discovered a couple of issues with NPI pieces. When the NPI is built, each piece is loaded up to the maximum size defined, regardless of the definition of primary and secondary quantities. Processes like page splits in the index still require some analysis and careful definition of free space. Both LOAD and REORG fill the pieces to their maximum size, so it imperative to have the space definitions match the piece size definition for proper DASD utilization.

We discovered another wonderful feature of type 2 indexes while constructing this system - the incredible ability of the index to handle a non-unique RID chain of over 600,000,000 duplicates (one key with that many duplicates). It not only works, it works well!

### ***Freespace***

When designing a table of this size and its indexes guessing at freespace can cost a lot of disk space. The PCTFREE and FREEPAGE becomes a BIG ISSUE! Due to the fact that the processing was not fully communicated prior to the start up of the production environment, the freespace was set, but had to be redefined soon after production processing started. Initially set to 10% PCTFREE and 0 FREEPAGE on both of the NPIs. After production started and we looked at how/where the inserts were occurring the PCTFREE was dropped to 5% and the FREEPAGE was adjusted. The smaller NPI had its FREEPAGE set 31 to keep the pages in the prefetch ranges but allow for more freepage because the data in this index was inserts in ‘clumps’. The other index would have inserts all over so the FREEPAGE set down to 0.

Freespace for the tablespace partitions was an interesting issue. The table was designed so that all data was inserted in date order in the last ‘active’ partition. Well that did not quite happen, due to the nature of where the data was coming from – it did not always make it the sequence that was anticipated by the table. So the last four partitions actually become our active

partitions allowing inserts and we also had to account for this in the freespace. All non-active partitions in the table had no freespace at all. The last four partitions now had to have FREEPAGE of 20 put on the last four partitions to allow for the non-sequential inserts occurring on this upper end so that we would not have to do reorgs on the partitions. The freespace was then removed when the partition became 'inactive'. I guess the lesson to be learned here is 'Designer Beware' - know your data and how it is processed.

### ***Loading the Data***

The following is an overview of the process of loading a large table, from converted data coming from one or more source systems. This process can be used anytime a large amount of data needs to be loaded and the process, if singular, would exceed all resources. The big effort here was also to make the process parallel. The trick was to allow all data coming out of conversion process to be archived since it might need to be input again to a load. Out of the process of writing the data to archive, each row of data for the table was passed to a sort routine, NOT TO SORT, but only to SPLIT into logical groupings of 125,000,000 rows each. These 125 million row file would large be split into separate 25 million row files for the actual loading into partitions holding 125 million rows. (Disk storage issues -- number of dataset issues -- etc.)

The overlap was to free disk and start each successive load process. This is a STEP-WISE PARALLEL PROCESS, freeing resources as it moves along. The first loads would use disk space from what would become the high end of the real table. You cannot normally have all the disk needed to hold the table and hold all the input data at the same time, plus the disk space to sort 6 billion rows, etc...

The first part was to take the 125 million row set and split and sort into the key ranges for each partition. The sorting of all this data has been deferred until this step. The sorted split output now is 5 separate load files on disk. Each set is archived to tape, then the original disk from the 125,000,000 row file is freed, and the next parallel load process is triggered to begin.

We deviated a little from the initial design for loading due to some physical constraints. The initial conversion processed 3.1 billion rows into approximately 40 individual loads processes then on into DB2 load-able datasets. The loads occurred in two groups: the first group loaded 3,135,772,036 rows and then updates were performed to a few previously loaded partitions for correction processing. Then the second group loaded 1,613,572,458 rows for a grand total of 4,749,344,494 billion rows. 3 loads were executed at a time with largest partitions averaging 1 hour each and were performed using third party load utility. At the same time, REORGs, RUNSTATS and IMAGE copies were performed. 3 partitions were copies in parallel, with largest averaging 12-15 minutes each with a third party utility.

When that data initially was loaded into the compression ratio of 38-48%. A reorg was then ran each partition in order to build a better compression dictionary. After the reorganization 99% compression ratio was achieved with an average of 37% page savings. These partition level reorgs were done before the NPIs were built. This necessary reorganization would also then have to occur for each 'new' partition when it data records were inserted. The fact that the NPIs would exist at that time would be come an issue.

### ***Building the Indexes***

The partitioned index was a duplicate, clustering index and was built during data load. The non-partitioned indexes were built after the data was loaded. The larger of the two indexes (78 pieces). (This index was actually also built after the first group was loaded so that some

updates could be performed, but was dropped before the rest of the data was loaded). The keys were unloaded prior to the build and this took approximately 4 hours for 4 partitions in 1 job, with 150,000,000 rows/job. There was 3 jobs executing in parallel on four partitions each. This took about 30 hours elapsed wall clock for unload/sort of index keys and then another 30 hours elapsed wall clock for index rebuild (recovery build phase). Then the smaller NPI was built taking up about 10 pieces and did not take nearly as long as the other to build.

### ***Maintaining the Data***

After the first load RUNSTATS was executed with 10% sampling across tablespace. Only after an entire tablespace runstats, could partition level runstats be taken because DB2 must collect statistics for the entire tablespace first. This execution took 84 hours elapsed and 45 hours CPU time. This was performed during the first 'test' building of this large table. The table was not 'dropped' but rather the data was 'deleted', so during final production load it was not necessary to run the RUNSTATS utility against the entire tablespace because the statistics still existed. Partition level runstats could then be used thereafter, and each partition runstats took approximately 30 minutes each. These partition level runstats were executed in tandem with the partition level image copies.

The partitioning index runstats took approximately 30 minutes per partition. The larger non-partitioning index took 10 hours for the entire runstats and the smaller averaging around 8 hours.

The Check Data to check RI and Table constraints after the load, was executed using a third-party Check Utility. The largest partitions averaged around 1 hour each with a total of 95 hours. However, it was determined that running the load with ENFORCE CONSTRAINTS was faster and this was performed during the production load. A check index for the largest NPI took 4 hours.

Reorganization of the non-partitioning Indexes was not possible as actual REORGs, due to size of required sorts. The indexes were recovered instead. In order to do this in a somewhat timely manner the unloaded key files were kept and no updates were allowed to NPIs. (This is a key point to make with NPIs on large tables...if they are designed and used properly there is not the maintenance issues everyone fears) Unloads were performed periodically on the last four partitions to keep the keys current. A "Reorg" of the larger NPI had elapsed time of 30 hours (providing the unload keys are current and available). For the partitioned index the reorg took average of 7-11 minutes per partition.

The tablespace partition reorganization did provides us with some fits. Each partition reorg took approximately 26-30 hours for a full (40mill) partition with vendor utility. This was painful because the actual reorganization of the data took only a couple of hours (2-3) but the rest of the time was spent updating the NPIs. We tried the IBM supplied Reorg it took approximately 17 hours. Even though the data was loaded in sequence for the most part, we still had to do REORGs on newly used partitions in order to get a better compression dictionary and then to also remove the freespace when the partition was no longer being inserted into. Version 7 promises to help improve the BUILD 2 phase of the reorg process in order to speed up the processing against the NPIs. It will do this by breaking up the NPI in several logical partitions and executing several tasks in parallel to do the updating of the NPI keys.

## ***Bufferpools***

So now there is a question about how do you size the buffer pools that have to support these large objects. Well, isn't there some rule of thumb that says to making the index bufferpool sized large enough to hold all non-leaf pages plus 20? Ok, so that's around 400,000 non-leaf pages for the three indexes even without the 20%. And what about the other rule of thumb that says to size a bufferpool large enough to hold the entire index? For the larger NPI on this table that is over 8 million pages. I don't think so! And even in V6 a dataspace would provide some relief, but these guidelines simply cannot work for these large objects. So basically we had to experiment and make the best of what we had. (which by the way about 400 meg of total real memory).

First we had to give a best guess at the bufferpool sizes. When we came up day one in production and the processing started against the large objects, we were able to measure the residency better for more appropriate sizing. After evaluating the processing it was determined how to better adjust all of the thresholds because our data was mainly being inserted and not referenced. So we decided to lower the bufferpool sizes and let the pages participate in a continual flow to disk and minimize I/O spikes. We also adjusted the system checkpoint interval as well. We then lowered the number of pages and adjusted the write thresholds. This worked very well for the small amount of memory we had and did not want to get into a heavy paging situation.

But the next issue was parallelism and having enough buffers to support it. We started looking into query parallelism and started to notice a good deal of fallback due to lack of buffers as well as observing that the degree of parallelism achieved at run time was often degraded from what was planned at bind time (found in the EXPLAIN output(degree determined at Bind) and Statistics Report (fallback due to lack of buffers) and the performance trace with IFCID 121 and 122 (showing the degree at run time)). This lead to us looking into what degrees had been chosen at bind time and it was discovered that all access against the large table where parallelism was applicable the degree was 88. How were we getting 88 degrees of parallelism at bind time when we only had 2 CPUs? Well, with CPU parallelism, anything is possible. We found that the 88 comes from how DB2 initially determines the degree at bind time. DB2 takes the total number of active pages in the entire table and divides this by the total number of active pages in the largest partition of the table. For our large table this was  $60655478/691126 = 88$ .

Fallback was high, overhead for fallback was high, and we had at tremendous problem tuning bufferpools for parallelism. Many bufferpools defined for random access(NPIs etc...), the VPSEQT set low and because the VPPSEQT is a percentage of VPSEQT – therefore no room for parallelism – more fallback problems. We tried inflating number of pages in bufferpools to try to support – not enough. Our final solution: we turned parallelism off because the system could not support it. We did worry that access against the large table would suffer but that was not the case. The applications were not suffering, but now the overhead was down.

## ***SQL and the Beast***

The system is now in production, and one might wonder about the impacts of SQL processing on a table that large. We were concerned with this also, but found that extensive use can be made of the enhancements of page range scanning. The major enhancement is the introduction of non-discrete partitions to be identified for scanning, but the actual scanning in the partitions is performed by an NPI. This is where the optimizer can identify the partitions to be processed: by comparing key data against the key ranges, and using the NPI as the access

method. Of course, this requires using a new technique in the SQL of allowing multiple OR conditions or an IN list predicates against the higher order columns in the clustering index; in the past, this was usually prevented since it caused a tablespace scan. As is usual with the growth of DB2, what was bad in the past now becomes good.

We also made use of key correlation statistics. The KEYCARD parameter on RUNSTATS indicates that cardinalities for each column, concatenated with all previous key columns, are to be collected. If KEYCARD is not specified then only FIRSTKEYCARD and FULLKEYCARD are collected. KEYCARD builds the frequency values for critical concatenated key columns - first and second columns, first, second and third columns. This was important for us because the large NPI often was only accessed using the first three columns of the five column index which would have resulted in a three column matching scan without knowing the extent of the scan. So by giving DB2 full key cardinality on all three columns with correlation statistics we obtained a very accurate filter factor.

SQL performance analysis should start as early as possible in the development life-cycle. We used exact index and table catalog statistics in the test and acceptance environments that were updated to mimic production for the purpose of validating the access paths. In addition to the statistics, the system DSNZPARMs, bufferpools, RID pool, EDM pool, and Sort pool were made to be the same in order for the optimizer to react consistently between these environments. If the environments had been inconsistent, the access path analysis is inaccurate, and we could have had some surprises in production.

Prior to production, the queries were reviewed for their adherence to best SQL access path selection. This was done by using an in-house developed system to run EXPLAINs against every program and analyze the access paths.

## Summary

In summary, it is safe to say that when it comes to building very large tables (VLTBs) that our design strategies must change and our access analysis must be more carefully thought out. You may not have billion row table yet, but in many organizations they are becoming more of a necessity and a reality. DB2 is ready to handle up to 16 terabytes in one table (not counting LOB columns – which can be up to 4000 terabytes per column), and I am sure that we will see more capacity increases in the future.

---

Susan Lawson is a Principal Consultant with YL&A. She is an internationally recognized consultant, teacher and lecturer specializing in database performance, VLDBs and data warehouses. She has been working with DB2 for since 1988 with a strong background in system and database administration. She was formerly an IBM Data Sharing advocate for the Santa Teresa Laboratory where she provided technical expertise for DB2 Data Sharing customers. Susan also has been published in several magazines such as 'IDUG Solutions Journal' and 'DB2 Magazine'. She is also a member IBM Database Gold consultants program and the IBM S/390 Gold consultants program. She has co-authored two books on DB2 and can be reached at [Susan\\_Lawson@YLAassoc.com](mailto:Susan_Lawson@YLAassoc.com).