

Index Design Analysis – why and how?

BY ROB LENZIE

T This article examines the analysis necessary to provide good index designs and describes some design and implementation considerations.

THE POWER OF THE INDEX

Indexing is the single most powerful tuning option available to a DB2 Database Administrator (DBA). No other DB2 system tuning facility can deliver the kind of performance improvements possible via a good set of indexes. Without the correct indexes, SQL statements will, at best, have to process a much higher number of DB2 pages to obtain the required answer set. At worst, a statement may perform a *Tablespace scan* or *Non-Matching index scan*. If the *Table scanned* is large and involved in a *join*, the statement could run for a very long time and place an intolerable load on the system.

Beneath all of the buzzwords and new technology architectures, users still need fast efficient access to their data. A batch, online or e-business system cannot tolerate inefficient access paths whatever the number and speed of the processors it is being run on. An inefficient access path consumes a significantly higher amount of finite CPU, IO bandwidth and buffer pool resources. If it is repeated many hundreds, thousands or millions of times in a day, consider the performance improvement and resource saving possible if the data request could be satisfied many times more efficiently via appropriate indexes.

Tablespace scans, *non-matching index scans* and other similar SQL performance problems are easy to spot and are generally resolved by performance analysts by SQL change or index modification. However, the most insidious performance problems are

those *moderately performing* high volume SQL, which with the correct indexes, could improve by a significant percentage, and which because of their volume over the life of an entire project, could save the project millions of dollars in excessive CPU consumption or unnecessary hardware upgrades. If optimum indexes were designed with the goal of optimizing the performance of the *entire workload* and not just fixing the *Tablespace scans* and *Non-Matching index scans*, the overall performance improvement would be undeniable.

WHY AUTOMATE THE DESIGN OF INDEXES?

Human resources

First, consider the fact that as mission critical application systems become more complex and ubiquitous, the demands placed upon very limited and expensive DB2 expertise becomes ever greater. Thus, as the limited specialist technical expertise becomes more stretched within an organization, the ability of these performance specialists to acquire and maintain the necessary detailed application knowledge required to be able to carry out effective tuning work becomes less and less.

Also, as hardware become cheaper, the proportionate cost of manpower in the total cost of ownership of computing resources becomes a much more significant factor. Companies need to maximize the return from their investment in technical expertise. Assigning highly skilled technicians to the complex and lengthy task of *Index Design and Tuning* becomes a less viable use of resources — particularly as the technology to automate this task becomes mature.

Packaged solutions

With the explosion of packaged ERP solutions, and their *tens-of-thousands* of tables and indexes, the ability of the DB2 performance specialist to take effective index tuning action becomes virtually impossible except in the most extreme of circumstances. Packaged vendors for their part, will endeavour to provide a reasonable set of indexes for these products, but each site's implementation and use of these tools will be quite different and thus require a site-specific index-design solution. Such solutions do not come about by themselves and require considerable time and expertise.

Change

Another factor in the equation is *change*. Even if a performance specialist was able to find the time to carry out a detailed tuning exercise and identify and design a set of optimal indexes for a particular workload, the application, which has been thus tuned, will inevitably change. These changes could be in the form of any combination of the following:

- Application changes
- User base may change
- Business may change as a result of a merger or takeover, and/or
- The applications concerned may be modified and fronted by a new Web front end or federated business transactions and suddenly be faced with millions of transactions per day

All of these change factors will give rise to changing indexing requirements.

continued on page 12

Thus, once having tuned a particular application to the best of their ability, the performance specialist needs to regularly revisit the application to reassess the indexes. Such work requires the dedicated time of the most expensive and scarce technical resources a company has.

New DB2 Versions

DB2 is a rapidly evolving technology. The DB2 Optimizer becomes stronger with each new release. What may have been a good index design for a particular workload may now be sub-optimal because of changes to the power and depth of perception of the DB2 Optimizer. How often do sites review their existing index designs in the light of new DB2 Optimizer features or new releases of DB2? Very few sites can afford to do so, because the manpower cost is so very high.

Complexity

With the growing complexity of many production application systems in use today, it is no longer possible, for even the most experienced of performance specialists to be able to both simultaneously perceive the diverse and conflicting indexing requirements of their systems, and then to translate these requirements into an *optimised set of indexes*.

Data volume

We have entered the *terabyte era*. With such inconceivably vast volumes of data, the performance implications of poor index design are devastating. No one can afford to inadvertently *scan a terabyte table, or even a portion of it*. Each access path needs to utilize the very best index(es) possible — or the end of the query may be when the operator finally cancels the job before completion, because it has consumed all available engines, for the last three hours of prime shift time! The *terabyte era* is where design mistakes can simply no longer hide. Index design errors are magnified a *thousand fold*, and may end up with SQL statements that run for weeks! If database designers have difficulties design-

ing indexes for *megabyte* and *gigabyte* tables, what chance do they have when tackling *terabyte* tables?

E-business

Index design is absolutely critical to e-business. E-business will hit all of the performance design buttons harder and faster than anything that has come before. It does not matter if the transactions are customer-driven Web-fronted applications, or *business-to-business* driven in the federated inter-business connections that are now being formed in the supply and delivery chains. E-business applications must be designed for extremely high volume and high performance. A slightly too long delay, and the customer will take their *clicks* and their business elsewhere. The applications concerned are likely to undergo rapid development and change. These applications may begin life as fairly simple transactions. However, the current pace of change on the Web will dictate that businesses will have to rapidly grow transactions that are very complex, whilst simultaneously sustaining ever increasing volumes of transactions.

The kind of hard earned application knowledge and expertise that underpins traditional approaches to index design are simply no longer viable on the e-business frontier. This world is moving too fast and the implications of poor performance are painfully and immediately visible to your customers.

AUTOMATING INDEX DESIGN - THE IMPOSSIBLE TASK

Before *index design methodologies* became available, there was no real index design process available to database designers. Index design was largely a *trial and error* and *hit and miss* affair. There were a number of rules of thumb, but mostly database designers relied on application experience and intuition for their index design decisions.

It should be clear to anyone who understands software development, that to get from a patchy set of rules to a fully functioning index design, which will find

an optimal set of indexes for any workload, is not a trivial task. Think about that for a moment. If you find it difficult to make the necessary index design decisions for your mission-critical applications, how is a dumb machine running some fancy algorithm going to do any better?

ESSENTIAL CHARACTERISTICS OF INDEX DESIGN

Variations of the steps outlined here need to be followed in some form by any of the index design tools now appearing on the market. These steps could also be followed manually by a diligent DB2 performance analyst wishing to assess or improve upon their existing index design.

SQL trace

The *de facto* start point for an *index design* effort must be the ability to capture a representative sample of live SQL. Without the ability to analyse and examine the characteristics of the SQL workload to be tuned, an index designer will never provide very useful indexes. One possible alternative approach would be to extract the available SQL from the DB2 catalog and use that as a start point. However, unless the tool can somehow consider the execution frequencies of the SQL making up the workload to be tuned, any indexes designed will be at best marginal and at worst useless.

Beware of any tool that does not use an SQL trace of the actual workload as a start point.

Analysis

Having captured some SQL, the next step is analysis of that SQL. Each SQL statement needs to be examined for *frequency of use, tables referenced, indexable predicates, select list columns, order by and group by clauses and column cardinalities*. The analysis process will provide information about which columns are worth considering in an index, and how these columns should be grouped together to form *single* and *multi-column* candidate indexes. The analysis phase can now proceed to attempt

to determine which are likely to be *good* candidate indexes. Extreme care needs to be taken in any decisions made here, as there is a tendency to end up throwing the baby out with the bath water. This is a very difficult problem to solve, as the number of candidate indexes for a moderately complex workload is extremely high. A typical start point is 1 x 1010 possible candidate indexes, from predicate analysis alone. No one can ever hope to evaluate such a large number of candidate indexes and complete in any reasonable time.

However there is much information to hand. Analysis can determine the *filter factor* and *SQL execution frequency* for each potentially indexable column. It can determine if a given column appears in a *Select list* and/or *Where Clause* and/or *Group By* and/or *Order By* clause. Similarly, it can determine if the column is the target of a *Join Predicate* and in exactly which SQL statements these *Join predicates* occur.

The application of some rules combined with the above information enables us to eliminate poorer performing candidate indexes without needing to evaluate them, whilst at the same time avoiding the elimination of good candidates.

Catalog statistics

The next phase, since the DB2 Optimizer is going to be used to help evaluate the indexes, is to generate some catalog statistics. This is probably the most critical part of the whole process. If the catalog statistics derived are not realistic, then any decisions made by the Optimizer based upon these statistics will be useless.

The best way to generate catalog statistics is to use a sample of live data. The candidate index is created and RUNSTATS run to collect the statistics. For OS/390, the value used for FIRSTKEYCARD is the live COLCARD value. For FULLKEYCARD the value obtained can be *scaled up*, and from this scaled up FULLKEYCARD value, an estimate for NLEAF and NLEVELS obtained. Finally the CLUSTERRATIO value obtained is used as is.

Candidate index evaluation

At this stage in the process, we have a number of *candidate indexes* for a given table. We have the SQL and their frequencies, which reference that table, and we now have catalog statistics for these *candidate indexes*.

There are a number of ways in which the DB2 Optimizer could help out:

Pecking Order: Create each candidate index in turn, populate the catalog with its statistics, EXPLAIN all of the SQL for the table, and check the PLAN_TABLE to see which, if any, of the SQL statements chose that index. Candidate indexes chosen by more expensive and/or frequently used SQL go to the top of the pecking order.

Slug it Out: Create some or all of the candidate indexes for a given table, populate the catalog with statistics and see which candidate indexes get chosen. Those chosen more often for the most expensive and frequent statements are the best candidates.

Cost Saving: First EXPLAIN the SQL for the table concerned, get the statement *cost* via the SQLCA or the DSN_STATEMENT_TABLE. Then *weight* each statements cost by execution frequency and then sum the weighted costs to get a *default_cost* for the table.

Next create each candidate index in turn, populate the catalog with its statistics, EXPLAIN all of the SQL for the table and get the statement cost with the candidate index in place, weight each statement cost by frequency and then sum by table to give the *with_candidate_cost*.

The candidate index cost saving is the *default_cost* minus the *with_candidate_cost*. Candidates with higher cost savings are in practice the most effective indexes.

The above techniques form the basis of how an optimizer based analysis design could theoretically evaluate candidate indexes. It is commonly agreed that *cost saving* is the most effective method.

Using the optimizer versus modeling approach to index selection

Some index designers have chosen to attempt to *second-guess* the DB2 Optimizer

by constructing a model of how they believe the Optimizer will view candidate indexes. This approach has a number of distinct advantages over techniques, which directly dialogue with the DB2 Optimizer. First, as intelligent as it is, the DB2 Optimizer is an expensive beast to use, particularly if it is being used to repeatedly EXPLAIN several hundred SQL statements with a large range of candidate indexes. Second, use of the DB2 OS/390 Optimizer requires the use of a mainframe class machine on which to run the optimization process. A model on the other hand can be run on a PC.

There are however serious disadvantages and considerations with modelling.

- First, if a model does the work, it is important that the model reflects the behaviour of the DB2 Optimizer. How well do the model developers understand the internals of the DB2 Optimizer, without direct access to the DB2 Optimizer developers and the DB2 code and algorithms?
- Second, how closely does *the model* follow the subtle changes made to DB2 from release to release, which usually completely redefine the optimisers character?
- Third, how can *the model* accurately reflect the behaviour of the DB2 Optimizer, if it does not have available to it a complete set of accurate *catalog statistics* for each of the *candidate indexes* being evaluated? Remember, accurate *catalog statistics* will only be available as a result of creating the candidate indexes and running RUNSTATS with some sample live data present, and collecting and *scaling up* the relevant statistic values.
- Finally, *models* can be very difficult and labour intensive to use. If the model requires the definition and manipulation of lots of parameters to enable meaningful results to be achieved, one has to question the models viability. If it takes as much work to get a result out of the model as it does to design the indexes manually, the benefits of using a model will be at best marginal from the index design perspective.

FINDING THE OPTIMUM SET OF INDEXES FOR A TABLE

It is one thing to devise a means of automatically evaluating candidate indexes using techniques such as described above. However, taking the step from being able to choose the **best** candidate index from a **set** of candidates, to determining the optimum set of indexes for a table, requires some quite different thinking and more sophisticated techniques.

SQL trace considerations

The sample of SQL traced must be representative of the workload for the indexes being designed. If traced SQL does not reflect the live workload then the candidate indexes evaluated may not be

the best ones. Also a key part of the process of arriving at a set of optimal indexes may involve the recommendation to remove some existing indexes. So it is critical that the SQL trace sample data include all important workload components, especially when the optimum set of indexes derived from this sample may suggest the removal of certain critical indexes. Be very careful and beware when removing any indexes.

SUMMARY

We've examined some of the analysis necessary to perform good index design and looked at some design and implementation techniques. This provides a flavour of processes used in the emerging index design technologies and provides a DB2 performance analyst with additional tech-

niques for assessing their index designs.

This paper is based upon Rob Lenzie's presentation "Grow your Own Perfect Indexes" presented at last year's IDUG conference in Dallas.

ABOUT THE AUTHOR

Rob Lenzie is a DB2 Gold Consultant and the UK DB2 Guide requirement secretary. He is a speaker at IDUG and other DB2 conferences on the subject of DB2 Index Optimization. He is currently architect of the EZ-XOP automated index design tool for the UK software developer Cogito. You can contact Rob at Rob.Lenzie@cogito.co.uk

sets. Thanks to our instancing/partitioning strategy discussed earlier, we avoid problems with DSMAX, since older instance data sets are typically inactive and not opened.

4 We use several hashing techniques for instancing and partitioning. For instance, in one system, we tinker with the actual store clock value a bit to achieve decent partitioning and still have ascending values to take advantage of Type 2 indexes. Also, we instance certain tables by criteria other than time (e.g., hash function on the external parcel label pointer table discussed earlier) to enable query access where time is not necessarily known.

5 We typically only run RUNSTATS one time when we have full volume. The way we have designed our database, we do not experience drastic differences in volumes. Whether a table has 5 million rows or

whether it has 7 million rows does not make a great difference to the optimizer or the chosen access paths. Running RUNSTATS on VLDB's can cause a great drain in resources. Once, when we had to make a structural change which involved an UNLOAD, DROP, CREATE, LOAD, RUNSTATS and REBIND, the RUNSTATS was the bottleneck as we could only run a few at a time.

Summary

We hope we have provided you some new ideas and techniques with the way we design and manage VLDB's at UPS. We're sure that there are some techniques you cannot adopt due to your own different business requirements. But, that is just the point with general rules of thumb! One size does not fit all in DB2!

The other point we hope we've made is that you cannot underestimate the

importance of design! Many DBMS maintenance tools on the market today are great and can save you a lot of resources, time and headache. But, in a VLDB, putting your effort into design up front on such things as key structures and purge processing pays great dividends at implementation!

ABOUT THE AUTHORS

Steven Di Spigna currently manages DB@ DBA's that support VLDB systems at United Parcel Service. He has 14 years of IS and DB@ experience. He can be reached at sdspigna@ups.com. Autumn Bonnabeau currently supports VLDB systems at United Parcel Service as a DB@ DBA. She has four years of IS and DB2 experience. She can be reached at abonnabeau@ups.com.