

Index Optimization White Paper

1 Index optimization theory and practice

This paper examines the need for Automated Index Design tools and describes some of the theoretical concepts, and implementation history leading to the current version of **EZ-XOP** (index-Optimizer) the *state-of-the-art* index design tool.



Technologies described in this Paper are Patent Pending and Either Owned by British Telecom and Licensed Exclusively to Cogito, or as in the Case of the Third Generation Techniques described herein are Owned by Cogito.

This paper is also included in the **EZ-XOP** User Guide, section 2 [Index optimization theory and practice](#).

Table of Contents

1	Index optimization theory and practice	i
1.1	The power of the index	i
1.2	Why automate the design of indexes?	ii
1.2.1	Human resources	ii
1.2.2	Packaged solutions	ii
1.2.3	Change	ii
1.2.4	New DB2 Versions	iii
1.2.5	Complexity	iii
1.2.6	Data volume	iii
1.2.7	e-Business	iii
1.3	Automating index design - the impossible task	iv
1.4	Essential characteristics of index design tools	iv
1.4.1	SQL trace	iv
1.4.2	Analysis	iv
1.4.3	Catalog statistics	v
1.4.4	Candidate index evaluation	v
1.4.5	Using the optimizer versus modelling approach to index selection	vi
1.5	Finding the optimum set of indexes for a table	vii
1.6	First generation technology - the genetic algorithm	vii
1.7	Second generation technology - heuristic algorithm	viii
1.8	Third generation technology – advanced heuristic algorithm with ratchet	viii
1.8.1	Third generation technology – advantages	xi
1.8.2	Third generation technology – the ratchet	xii

1.1 The power of the index

Indexing is the single most powerful tuning tool 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 will *Tablespace Scan*, or *Non-Matching Index Scan*. If the *Tablespace Scan* is large and involved in a *join*, the statement could run for days 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, on-line 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 proportion of finite CPU, IO Bandwidth and Bufferpool. If it is repeated many hundreds, thousands or millions of times in a day, or if it is part of a *Multi-Table Join*, 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 glaringly obvious 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. However, the nature of this insidious SQL is very much like the nature of an iceberg. The bit that shows above the water; the *Tablespace Scans* and *Non-Matching Index Scans* are glaringly obvious and can be fixed very quickly with minimal effort. However, the 90% of the iceberg that is *below* the water, what we call the *Insidious SQL*, is not so easy to deal with. The *payback* is generally less, and the effort to get the *payback* is proportionately much greater. However, 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.

1.2 Why automate the design of indexes?

1.2.1 Human resources

Firstly, 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 get as big a return from their investment in technical expertise as they can. Assigning highly skilled technicians to the complex and lengthy task of *Index Design and Tuning* becomes less and less of a viable use of resource, particularly as the technology to automate this task is now becoming mature.

1.2.2 Packaged solutions

With the explosion of packaged solutions, particularly the likes of SAP and PeopleSoft, with 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.

1.2.3 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 change. These changes could be in the form of any combination of the following : -

- application changes
- the user base may change
- the way the users use a particular application may change with time in response to business process changes
- the business may change as a result of a merger or takeover
- the applications concerned may now be fronted by a 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. 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. This is so as to ensure that over time, the index design is changing as well to meet the changing needs of the workload, thereby enabling the application to continue to deliver optimal performance. Such work requires the dedicated time of the most expensive and scarce technical resources a company has.

1.2.4 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 releases of DB2 ? Very few sites can afford to do so, because the manpower cost is so very high.

1.2.5 Complexity

With the growing complexity of many production application systems in use today, it is no longer possible, if indeed it ever was 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 convert these requirements into an *optimized set of indexes* which delivers maximum throughput for least cost.

1.2.6 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 *Tablespace Scan* a *terabyte table*. Each access path needs to go in there with the very best index possible. Or the end of the query may be when the operator finally cancels the job, because it has already consumed all 16 engines, for the last 3 hours of prime shift time! The *terabyte era* is where design mistakes can simply no longer hide. Index design errors are magnified thousandfold, and may end up with SQL statements that run for weeks! If database designers have difficulties designing Indexes for *megabyte* and *gigabyte* tables,, what chance do they have when tackling *terabyte* tables?

1.2.7 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 a 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 extremely high volumes of transactions.

If business cannot meet these challenges, and support the *sky-high* transaction volumes, whilst delivering acceptable performance to their end customers, then in this new e-Business battlefield, those failing will face ruthless elimination, bankruptcy and closure at the hands of their competitors.

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 too painfully and immediately visible to your most precious asset, your customers.

1.3 Automating index design - the impossible task

Before *automated index design tools* became available, there was no real index design methodology 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 tool, 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?

1.4 Essential characteristics of index design tools

1.4.1 SQL trace

The *de facto* start point for an *index design tool* 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 design tool 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.

1.4.2 Analysis

Having captured some SQL the next step is analysis of that SQL. A viable tool will need to examine each SQL statement 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 the tool with 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 for tools to end up "throwing the baby out with the bath water". This is a very difficult problem for any tool to solve, as the number of candidate indexes for a moderately complex workload is extremely high. **A typical**

start point is 1×10^{10} possible candidate indexes, from predicate analysis alone. No tool 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. **XOP** can determine the *filter factor* and *SQL execution frequency* for each potentially indexable column. It knows if a given column appears in a *Select list* and/or *Where Clause* and/or *Group By* and/or *Order By* clause. It knows if the column is the target of a *Join Predicate* and it knows in exactly which SQL statements these *Join predicates* occur.

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

1.4.3 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.



The **EZ-XOP** product includes tools to extract sample data from the live system to be loaded to the XOP system.



A not so good way of generating catalog statistics (used by other tools) is to attempt to guesstimate values without a live data sample. It is possible to get a very approximate value for FULLKEYCARD using certain communication theory algorithms, however it is impossible get a good value for CLUSTERRATIO without recourse to data. Since CLUSTERRATIO is one of the most critical index statistics, the value of any tool which relies on guesstimates for this metric will always be marginal. Recall that CLUSTERRATIO is a measure of how closely the ordering of rows in the table matches the ordering of rows in the index concerned. The purpose of CLUSTERRATIO is to give the optimizer a feel for how expensive row retrieval via a particular index will be. High CLUSTERRATIO values mean that there will be many rows got per data page read and that each row will probably be read only once. A low CLUSTERRATIO value will mean that at worst there will only be a single row per data page and that if the index scan is large and the BUFFERPOOL saturated, a given data page may be read many times. Attempting to guess a number for this performance critical metric, when the value is wholly data dependant, is going to give any tool that does so an extremely hard time providing the user with the kind of high performing indexes they need.

1.4.4 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 an automated index design tool *could* get the optimizer to 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. Bear in mind that DB2 is likely to crash if an attempt is made to create more than 900 indexes on a single table, should you ever try ! Also with large numbers of candidate indexes present, the probability of SQLCODE -101 – THE STATEMENT IS TOO LONG OR TOO COMPLEX goes up dramatically.
- **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 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 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 automated index design tool could theoretically evaluate candidate indexes. It is commonly agreed that cost saving is the most effective method and it is in fact a very advanced form of cost saving methodology that is used by EZ-XOP.

1.4.5 Using the optimizer versus modelling approach to index selection

Some vendors 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. Firstly, 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. Secondly, 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, since the work is done by a *model*, 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 optimizer's 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 model's 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.

1.5 Finding the optimum set of indexes for a table

It is one thing to devise a means of automatically evaluating candidate indexes using any of the above techniques. 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 approaches.

The methodology used by the EZ-XOP product has evolved over a period of over six years of development and several quantum leaps in terms of sophistication. The history is described here in order that users appreciate the amount of R&D which has gone into this product. It is extremely unlikely that any other available index design tool uses techniques which are as well developed or as effective.

The techniques used by EZ-XOP and described here are covered by patents either owned by British Telecommunication (BT) PLC and licenced exclusively to Cogito, or owned outright by Cogito.

1.6 First generation technology - the genetic algorithm

EZ-XOP started life as a *genetic algorithm* (GA). *Genetic algorithms* are a class of search algorithms especially good for finding or approaching an optimal solution in a very large search space, using the Darwinian principles of *survival of the fittest*, *sexual recombination* and *mutation*.

The algorithm starts its first generation, with a population of random solutions, which are each represented as a genome, consisting of genes. Each member of the population is evaluated and scored using a *fitness_function*, which is a measure of how good that particular member is. In XOP terms we used *Cost Saving* as described above as the *fitness_function*.

Once all of the initial population has been scored, the subsequent generation is built, using the higher scoring members of the earlier population. The lower scoring members are eliminated. The new generation is built using the GA techniques called *crossover*, *mutation* and *elitism*. *Crossover* involves taking two parents from the earlier population and combining part of the genes of one parent with remaining genes of the other parent to produce a child, which is then mutated. *Mutation* involves making random changes to the child. An alternative to crossover is mutation used on its own. Here a child is produced by taking a single parent from the earlier generation and mutating the parent to produce a child. Lastly, *elitism* takes the best member from the earlier generation and propagates it unchanged into the new generation. This ensures that the most fit member of the earlier generation always finds its way into the following generation – *survival of the fittest*.

The idea was good, but the process was very slow to run. It took many hundreds of generations to get anywhere close to a good solution and the *catalog statistics* for each *candidate index* being evaluated needed to be generated before the GA started. This could take many days alone.

1.7 Second generation technology - heuristic algorithm

Since we had to devise a way of scoring each design in the GA, we already had a means for reliably scoring indexes. This used the *Cost Saving* technique described in *Candidate index evaluation* above. Now it just so happened that by examining indexes in descending order of cost saving, the technique invariably put what an experienced DBA would consider to be the best index at the top of the list. This is not that surprising, since this technique is exploiting the power of the DB2 cost based optimization technology, to get the optimizer to tell us which of a set of *candidate indexes* it likes best. It is a bit like putting on DB2 optimizer glasses and looking at each candidate index through these optimizer glasses! Since the DB2 optimizer is very smart, it invariably picks the best index.

After much practical and conceptual experimentation with this *Cost Saving* technique, where many different ideas and techniques were considered, a stroke of inspiration suggested that the technique could be used iteratively to come up with an optimum *set of indexes*.

Consider this. The start point of the process is what we call *default indexes*. These are the *Primary*, *Clustering* and *Foreign Key indexes*. Then we evaluate each of the set of *candidate indexes* for a particular table. We create each *candidate* in turn and get its *Cost Saving*. We can now say which is best of the initial *Candidate indexes*. Nothing new so far.

Here's the trick. If we now add the best *Candidate Index* to the set of *Default Indexes* and then re-evaluate all of the remaining *Candidate Indexes* again, we will get a quite different picture of *Cost Saving*. Those *Candidate Indexes* similar to the *now Default Candidate Index*, will all now have a very low or zero *Cost Saving*. Whilst *Candidate indexes* dissimilar to the *now Default Candidate Index*, still have considerable *Cost Saving* potential. What we do now is to now pick the best *Candidate Index* again and add that to the set of *Default Indexes*. Thus the process repeats until there is no more *Cost Saving* to be had.

What this technique does is build on the original *Cost Saving* technique, by once again getting the DB2 optimizer to tell us which of the *Candidate Indexes* at each phase of processing it likes best.

Now there were some problems with this algorithm, which is why it never saw the light of day in a commercial product. If the number of *Candidate Indexes* was large, and for a workload with any degree of complexity it would be large, then the run time of the optimization engine could easily be counted in many days. Also because of the nature of the algorithm, *Catalog Statistics* for each of the *Candidate Indexes* was required before processing could begin. *Catalog Statistics* generation would also require several days of elapsed time.

1.8 Third generation technology – advanced heuristic algorithm with ratchet

The underlying principle of the 2nd generation technique was sound, but elapsed time was the major problem. People would not buy technology that ran for days or weeks, however good the indexes it designed might be.

We experimented with ways of eliminating *candidate indexes* and pruning the workload SQL. Both of these techniques had a dramatic effect upon elapsed time – but it was still measured in tens of hours or days. Also, there was always the nagging doubt, that however good the elimination techniques were, that we might inadvertently be eliminating good *candidate indexes* or a particularly critical SQL statement. Something quite different was needed.

These reservations are finally resolved in the current 3rd generation technique, described as advanced heuristic algorithm with ratchet. The technique starts from the *ground up* and *grows* indexes, much like the way crystals grow. This is a radical diversion from the earlier work which had focussed on techniques for eliminating candidate Indexes. Early prototypes proved that the new process would grow

indexes that were very similar to those produced by the second generation algorithm, but now in a couple of hours, rather than the days required by the earlier algorithm.

The following illustrates how the process works with a simple example.

Suppose we have a simple workload from which we have identified the following column groupings, which we call *Predicate Sets*, from which we wish to grow a set of Indexes :

A, D, E, G
B, E, G

We start the process off by evaluating in turn each of the possible single column indexes :

A ASC
A DESC
B ASC
B DESC
D ASC
D DESC
E ASC
E DESC
G ASC
G DESC

Suppose we choose A ASC as the best index and add that to the list of *default indexes* for the table. We now evaluate the following pairs of single indexes :

A ASC	B ASC
A ASC	B DESC
A ASC	D ASC
A ASC	D DESC
A ASC	E ASC
A ASC	E DESC
A ASC	G ASC
A ASC	G DESC

We now drop A ASC and evaluate in turn each of the following two column indexes :

A ASC, D ASC
A ASC, D DESC

A ASC, E ASC
A ASC, E DESC

A ASC, G ASC
A ASC, G DESC

Suppose we now choose (A ASC, D DESC) as the best index and add this to the list of *default indexes* for the table, eliminating the earlier (A ASC) which this new index supersedes. We now evaluate the two column index (A ASC, D DESC) along with each of the following single column indexes

A ASC, D DESC	A ASC
---------------	-------

EZ-XOP
Index Optimization White Paper

A ASC, D DESC	A DESC
A ASC, D DESC A ASC, D DESC	B ASC B DESC
A ASC, D DESC A ASC, D DESC	D ASC D DESC
A ASC, D DESC A ASC, D DESC	E ASC E DESC
A ASC, D DESC A ASC, D DESC	G ASC G DESC

We also evaluate the possible three column indexes determined by the above predicate sets :

A ASC, D DESC, E ASC A ASC, D DESC, E DESC
A ASC, D DESC, G ASC A ASC, D DESC, G DESC

Suppose we now choose the two indexes (A ASC, D DESC) & (B ASC). These indexes are added to the list of chosen indexes, replacing any superseded indexes where necessary. We now evaluate the following groups of three indexes :

A ASC, D DESC	B ASC	B DESC
A ASC, D DESC	B ASC	D ASC
A ASC, D DESC	B ASC	D DESC
A ASC, D DESC	B ASC	E ASC
A ASC, D DESC	B ASC	E DESC
A ASC, D DESC	B ASC	G ASC
A ASC, D DESC	B ASC	G DESC

We also evaluate the groups of two indexes made up of (A ASC, D DESC) and the set of possible indexes with B ASC as the first column :

A ASC, D DESC	B ASC, E ASC
A ASC, D DESC	B ASC, E DESC
A ASC, D DESC	B ASC, G ASC
A ASC, D DESC	B ASC, G DESC

And finally we also evaluate the groups of two indexes made up of B ASC and the set of possible indexes with A ASC, D DESC as the first and second columns :

A ASC, D DESC, E ASC	B ASC
----------------------	-------

A ASC, D DESC, E DESC	B ASC
A ASC, D DESC, G ASC	B ASC
A ASC, D DESC, G DESC	B ASC

Now we choose the two, two column indexes (A ASC, D DESC) and (B ASC, E ASC) as the best index. These indexes are added to the list of chosen Indexes, replacing any superseded indexes where necessary.

And so the process goes on...

1.8.1 Third generation technology – advantages

The index design grows its way into the workload, matching the subtle characteristics of the workload requirements as it goes, guided only by *Cost Saving*. Like a *key growing into a lock*.

The advantages of this process over earlier generations of the technology are many fold.

- This is a very elegant and deeply satisfying solution. The indexes *grow like crystals*, bedding themselves down into areas of the workload providing the greatest *Cost Savings*.
- There are very few indexes to evaluate with this technique. The start point is the set of single column indexes derived from the *predicate sets*. As more and more columns get chosen and added to the existing index structures, each step in the process reduces the search space still further until there are either no more *Candidate Indexes* to evaluate, or there is no more *Cost Savings* available. The route for the search is directly through the highest *Cost Saving* space of the workload.
- At each step in the process, the *fittest* column survives and either forms a new index or an additional column to an already existing index. Everything else is eliminated.
- There is no longer any need for potentially disastrous *candidate index elimination* techniques or *SQL elimination* because the process is now extremely quick and ruthlessly efficient.
- *Catalog Statistics* are generated for each set of indexes which will be evaluated in the next phase. There is no requirement for several days of processing time to derive the necessary *Catalog Statistics* for the *Candidate Indexes*. This means that much larger *Live Data samples* can now easily be used, thus providing more accurate *Catalog Statistics* to the process, which will mean a more finely tuned set of optimal indexes.
- This technique automatically grows sets of indexes if there is a need for more than one index. If there is only a requirement for a single index, then that is all that will grow. Remember *Cost Saving* is the guiding force here. If the process can get a higher *Cost Saving* from multiple indexes then it will.
- The process can work with single SQL statements or complex SQL workloads consisting of thousands of distinct SQL statements.
- The process is many orders of magnitude faster than the earlier generation XOP technology.
- The process evolves a set of indexes that are very similar to that produced by the earlier generation XOP technology. The difference is that with the new technology, the indexes for a complete workload can now evolved in a couple of hours at most, instead of several days.

1.8.2 Third generation technology – the ratchet

The third generation algorithm as presented up until now has a minor limitation. The process tends to favour *high cardinality* columns as the building blocks for *Candidate Indexes*. Now, the experienced DBA may be aware that sometimes the best indexes might be made up of two or more *low cardinality* columns.

Now a characteristic pattern of this *third generation technology* is that the *Cost Savings* start out large and very rapidly diminish. However whilst running the engine, late on in processing, it is apparent that indexes would appear made up of *Low Cardinality* columns that produced a characteristically large increase in *Cost Saving*, showing up as a big blip in the tailing off *Cost Saving* curve.

An idea suggested itself, that if we rolled back the process in time, until the most recent *Cost Saving* was greater than the blip, inserted the *Low Cardinality* column index there, and deleted any *Chosen Indexes* that followed, and allowed the process to continue, the process would capture this *Low Cardinality* column index into the bedrock of the index design being evolved. So *the ratchet* was born!

Essentially, the ratchet allows late on emerging higher *Cost Saving* indexes made up of *Low Cardinality* columns to be included in the evolution process earlier on. This has the effect of biasing the process more towards greater overall *Cost Saving* for less index space.



Note that much of the early prototype design work was conducted at British Telecommunication PLC, and that all of the techniques described are either covered by patents owned by British Telecommunication PLC and licensed exclusively to Cogito Limited, or patents owned by Cogito Limited.

The concepts and procedures described in this document have been implemented in the **EZ-XOP** (index **OPT**imizer) product developed and marketed by Cogito Limited, under licence from British Telecommunications PLC.

For further information Contact:-



Cogito Limited
Chatham House
Goshill Road
Chislehurst
Kent BR7 5NS
UK

Tel: +44 (0) 208 295 1970

Fax: +44 (0) 208 195 1967

email: software@cogito.co.uk

URL: www.cogito.co.uk

Visit the Cogito web site for details of worldwide distributors.



In USA and Canada

TACT Software Inc
84 West Park Place
Stamford, CT 06901

Tel: (203) 967-8228

Fax: (203) 967-1122

email: software@tact.com

URL: www.tactsoftware.com