**Plan Bouquet based Techniques for Variable Sized Databases**

Achint Chaudhary

April

April 10th, 2020

MTech Project Report (CSA)

**0 Abstract**

OLAP applications often require a certain set of canned queries to be fired on database with the varying constants in queries. For optimal execution of these queries, query optimizer does select a strategy known as the query execution plan. These choices are based on cardinality estimates of various predicates that often hugely differ from actual cardinality values encountered during execution. Due to this reason, optimizer choice leads to high inflation in actual execution cost as compared to predicted cost during optimization.  
  
An altogether different approach for query processing was proposed in 2014, named Plan Bouquet. Basis of which is selectivity discovery at run-time by repeated cost bounded execution of carefully chosen to set of plans. This technique provides strong bounds independent of data distribution.

However, Plan Bouquet on cost sub-optimality is not designed to be robust against large updates in the database. This work focuses on observing limits up to which size of database can be increased without serious deterioration in performance guarantee. Also, we will provide incremental algorithms that can use information from plan bouquet compiled in past and extend it, to provide further robust execution without incurring overhead of re-compiling entire plan bouquet.

**Sec 1 Introduction**

Database query optimizer chooses a plan covering various structural choices of logical and physical operators for query execution. These choices are based on the cost of each operator which is calculated using number of tuples it will process known as cardinality. Cardinality normalized in range of [0, 1] is known as selectivity throughout literature.

These selectivity values are estimated before query execution based on some statistical models used in classical cost-based optimizers. An entirely different approach based on run-time selectivity discovery is proposed called Plan Bouquet, which provides for the first time strong theoretical bounds on worst-case performance as compared to optimal performance possible from all the available plan choices.

For each given query, predicates prone to selectivity error contribute as a dimension in Error-prone Selectivity Space (ESS). The set of optimal plans over the entire range of selectivity values in ESS is called Parametric Optimal Set of Plans (POSP). POSP is generated by asking optimizer's chose plans at various selectivity locations in ESS using Selectivity injection module. Cost surface generated over entire ESS is called Optimal Cost Surfaces (OCS). An Iso-cost surface is collection of all points from OCS which have same optimal plan’s cost at these points.

A subset of POSP is identified as Plan bouquet, which is obtained by the intersection of plans trajectory with OCS, creating multiple Iso-cost surfaces, each of which is placed at some ratio proportion (r\_pb) of cost from the previous surface.

Since each plan on an iso-cost surface has a bounded execution limit, and incurred cost by execution using bouquet will form geometric progress. The figure below shows the performance of 1D plan bouquet w.r.t to optimal oracle performance.

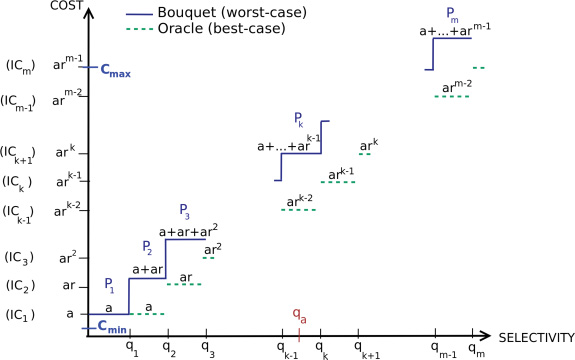


Fig 1. Cost incurred by Oracular vs Bouquet Execution

In above figure, various plans up to actual selectivity value are executed. Each plan has a limit provided by the next iso-cost surface. This yields total execution cost of

This value is minimized using r\_pb=2, which provides theoretical worst case bound of 4 times the optimal execution time. Extending the same idea to multiple dimensional ESS, MSO guarantee will become 4\*p, where p is maximum cardinality of plans on any of iso-cost surface.

Since computing value of p will need huge compile time effort, it is platform dependent and also low value of p is desired, which is dependent & obtained using Anorexic reduction heuristic at the time plan bouquet was developed.

Later a improved algorithm called SpillBound is invented, which is able to provide performance guarantee based only on query inspection and is quadratic function in number of error-prone predicates, which is same as dimensionality of ESS. MSO guarantee obtained by Spillbound is

D\*\*2+3\*D

We will be using Spillbound in some sections for our work, as it provides pre-compilation performance guarantee which are platform independent in nature, just from query inspection

**2. PROBLEM FORMULATION**

2.a Notations

2. b Assumptions

**Plan Cost Monotonicity (PCM)**

This assumption implies that if location b spatially dominates location a in ESS, Cost of optimal plan at location b is more than cost of optimal plan at location a.

(b>a) 🡪 Cost(b) > Cost(a)

This also comes from a simple fact that processing more tuples will incur more cost

**Axis Parallel Concave (APC)**

This assumption is on Plan Cost Function (PCF) which is not just monotonic but exhibit a weak form of concavity in their cost trajectories. For 1D ESS, PCF is said to be concave if for any two selectivities locations x1, x2 from ESS and any w in [0,1] below condition holds

F(w\*a + (1-w)\*b ) >= w\*F(a) + (1-w)\*F(b)

Generalizing to D dimensions, a PCF is said to be Axis Parallel Concave (APC) if the function is concave along every axis-parallel 1D segment of ESS.

Which simply states, that each PCF should be concave along every vertical and horizontal line in the ESS

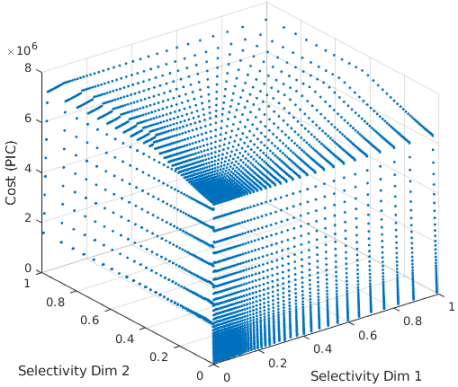
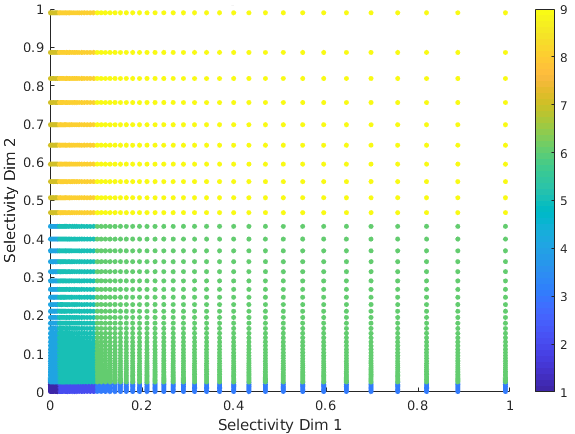
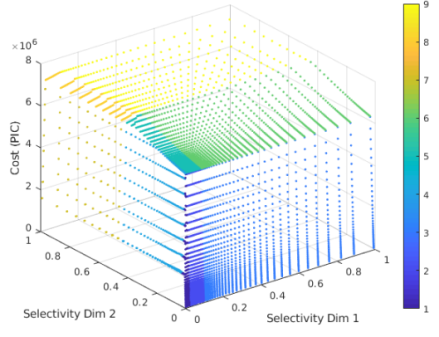
Further, an important and easily provable implication of the PCFs exhibiting APC is that the corresponding OCS, which is the infimum of the PCFs, also satisfies APC. Finally, for ease of presentation, we will generically use concavity to mean APC in the remainder of this work.

**Bounded Cost Growth (BCG)**

Similar assumption is made by [SUDARSHAN], and have proved that in practice slope of plan cost functions and OCS are bounded in form of a polynomial function in chance of selectivity. They have claimed that identity function f(a)=a suffice in practice

**Piecewise Axis Parallel Linear (APL)**

Plan Cost functions & OCS are shown to be piecewise linear in [HS02]. This property commonly comes from the fact that partial derivatives of common physical operators (except the sort operator, which is seldom found in industry strength benchmark) are linear in nature. We will use this property for design of various algorithms and improvements in existing methods.

When it is the case that OCS or Plan cost functions are not truly piecewise linear, a coarse approximation of piecewise linear function can still be fitted to them. Similar work is done in our lab also in past by [SANKET]. While in our work don’t explicitly need to fit any such piecewise linear function. This will reduce our effort of fitting points from entire OCS into a piecewise APL function

**EPP are the only predicates**

For present analysis, we have considered that all of query predicates are Error-prone, there is also no trivial predicate, which means each relation has some filter predicate applied over it, which will be also considered as error-prone in our conservative assumption.

Rational behind this assumption is that, if we supply same selectivity value to both old and new database, their outputs will be of different cardinalities. If we have not gone with this conservative assumption, we will need to determine change of scale for AKP and TP, and in that case, maybe re-compilation of bouquet may be needed. Removing this assumption and detecting is a problem we will later try to encounter

**Perfect Cost Model of Optimizer**

This assumption states that poor choices of plan come only from cardinality estimation error of optimizer and not from cost model itself. While, we have assumed perfect cost model of optimizer, an optimizer with bounded cost model will also work well. Improving, which is an orthogonal problem, one work on static tuning, proves most actual cost values to lie in 30% of estimated cost values after tuning.

**Selectivity Independence**

We assume that selectivity of predicates are independent of each other, while this is a common assumption in query optimization literature, it often does not hold in practice.

2.c Performance Metrics

MSO

ASO

2.d Selectivity Intervals for study

Selectivity is the fraction of tuples out of maximum possible tuples, that can come out of a query predicate. Notation of selectivity is devised to make study of ESS independent of cardinality values. This has motivated in past literature that selectivity value be always bounded in range [0,1].

Looking at type of changes in database, either it can be change in data distribution or change in volume itself. Since plan bouquet and later devised techniques are robust to distributional changes, if database has only gone through distribution change, there will be no need for re-compilation, under assumption that all predicates are error prone.

Considering the case of increase in size of database, Number of tuples to process and maximum possible tuples for any predicate generally increase. So, if we go with [0,1] picture of selectivity for both earlier & updated instance, same selectivity value will result in different cardinality on database before and after volumetric update.

Since, query optimizers are cost based where cardinality is taken into account for cost calculations. Also, we will look into to re-use iso-cost contours drawn in past. Points on any contour all has same cost, which comes from cardinalities and cost model. So we will chose to keep uniform selectivity to cardinality mapping across multiple sized databases.

In a loose sense, selectivity to cardinality mapping should not be changed across different sized instances. For example if total tuples in a relation before update are 100, and post update it has become 200. Then, 0.5 selectivity on both databases should return 50 tuples. So for initial database selectivity values are from [0,1], while for instance post update, legal selectivity interval is [0,2]. This may seem to be fuzzy at first, but we will later show importance of this notation. Only the base instance on which database is first compiled has selectivity values in [0,1]

Also we will look change in volume from ration of change in maximum cardinality out of a predicate, since each axis of ESS denotes a predicate. So ESS upon change in database may have grown differently on each axis of ESS. Below example pictorially show this approach and potential regions, for which Incremental algorithms are needed to be developed.





In above diagram. Green region is the portion for which standard or optimized compilation procedure will be required. For now we need algorithms to

1. Extend contours from Red region, which are intersection on boundaries between red & green regions
2. Check modification in cost of contours and their points lying completely in red regions, which will be helpful to provide relaxed MSO guarantee from using old contours and plans from red region
3. Incrementally compile segments of contour in red regions, if usage of old contours are deteriorating performance guarantee significantly

**3. CHALLENGES IN HANDLING DATABASE UPDATE**

Plan bouquet is suitable for canned queries as compilation overhead of entire ESS enumeration will take O(RES\*\*Dim) cost, which is amortized over repeated invocations. Under situation of change of database size, placements of ideal contours can be totally different from existing contours built for old database size, which may result that old bouquet plans are totally different from new bouquet contours. Under above mentioned conditions it seems that a re-compilation will be needed.

Now we will look at how compilation (iso-cost surface identification) in past work is done, and why under all present options compilation at it first place is resource exhaustive.

Origin & Terminus are the extreme ends of ESS, at which optimal costs are obtained via getting optimal plan using selectivity injection. These cost values are called C\_min, C\_max respectively, using these two values, number of contours (m) is obtained as follows.

M = ceil( log( (C\_max/C\_min) , [base=r\_pb] ) ) + 1

These M iso-cost contours are to be drawn at r\_pb cost ratio successively. Now next part is to detect points from ESS & optimal plans at those locations, lying on each of the contour. For now there are two options available for contour construction Full ESS enumeration and NEXUS. Let’s see them one by one

**Full discretized ESS Enumeration**

This is a naïve approach, in which optimal plan and its cost at all points of ESS is asked from query optimizer. Points at which optimal plan’s cost is equal to any cost value of iso-cost contour is qualified to be added to that contour. This will incur O(RES\*\*Dim) cost. Where RES is resolution chosen to discretize ESS, while Dim is dimension of ESS. Each dimension in ESS represents a error-prone predicate.

This approach is certainly Exponential in Dim, and also a suitable value of RES should be chosen to make overall cost computationally feasible. Full space enumeration can totally exploit parallel architecture of modem multi-core systems available.

**NEXUS (NEighborhood Exploration Using Seed)**

An optimization over full space enumeration is introduced with plan bouquet. NEXUS is an algorithm proposed to avoid making unnecessary optimizer calls to points lying in between contours. If we have total M iso-cost contours to discover, worst case complexity of NEXUS can go up to O( M\*D\*(RES\*\*(D-1)) )

At first NEXUS seems to be promising for reducing compilation overhead, but faces multiple following issues (as mentioned by [SRINIVAS]):

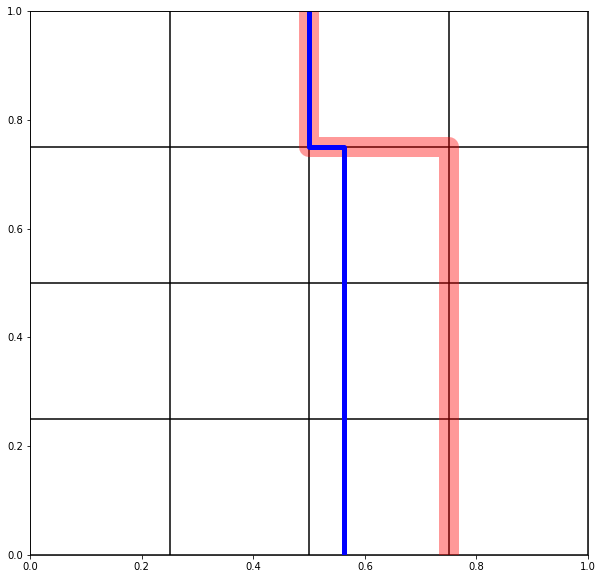
1. If large number of contours needs to be drawn, NEXUS is effectively close to Full space enumeration, especially in high dimensional ESS
2. If a lower bound is known on query selectivity, Spillbound is able to shrink ESS, while NEXUS needs redrawing of all contours from scratch

(3) Randomized contour placement to introduce fairness in Plan bouquet needs more contour need to be drawn. This makes NEXUS cumulatively more expensive than full space exploration

Also, both above methods of finding iso-cost contours make a common assumption that, Resolution of discretized ESS grid should be sufficiently high such that we can always find contiguous iso-cost locations even with small values of a, say, 0.05.

Next we will see, some issues which are common to both Full ESS exploration and NEXUS

ISSUE WITH USING LOW RESOLUTION FOR HIGH DIMENSIONAL ESS



To empirically avoid above explained issue, when working with low resolution and high dimensional ESS and also make things computationally feasible, geometric distribution of selectivity value was used on each axis in practice

This use of low resolution and geometric distribution is never explicitly stated in literature, and may violate MSO guarantees in practice. Which is not observed till yet, but proof for the same is also pending like a conjecture. Rational till now for use of Geometric distribution is that it captures many points in low selectivity values, and most changes in plan choices takes place in low selectivity values.

**4. OUR CONTRIBUTION**

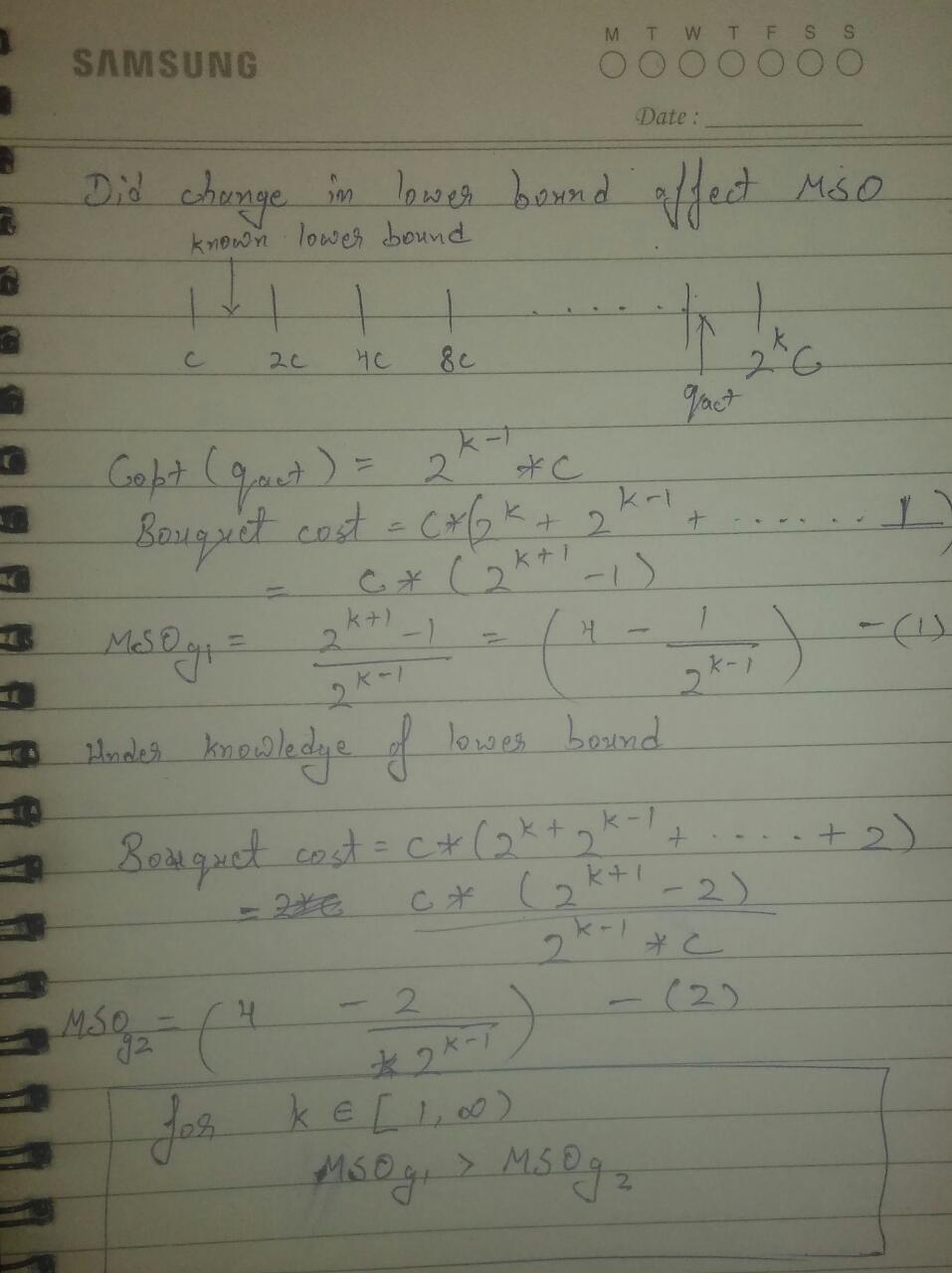
**4.a Complexity of NEXUS & Scope of improvement (Reducing overall compilation efforts)**

NEXUS as discussed in [SRINIVAS THESIS] faces multiple issues, two major of these are:

1. Same effective cost as full ESS enumeration
2. Need to redraw contours from scratch if lower bound on any selectivity predicate is known.

Before we start working on improved version of nexus, We will first try to look at second argument made against NEXUS

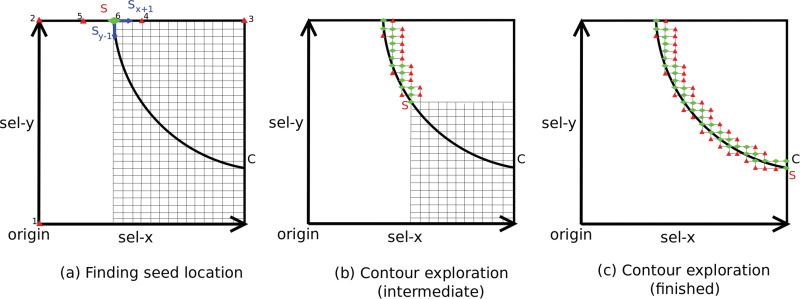
Impact of known lower bound



Since, now we are clear that knowledge of lower bound does not impact either of NEXUS or Full Space enumeration.

Now we will look at cost aspects incurred by NEXUS and its effective similarity as Full ESS Enumeration.

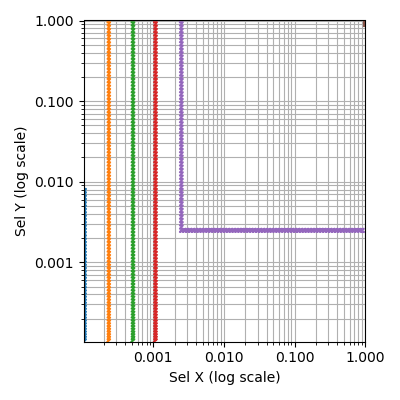
Base idea of NEXUS is to first locate a seed, which is one end point of contour, and then use it to discover rest of the points of contour lying in its neighborhood. Below example borrowed from [ANSHUMAN PAPER] shows working of NEXUS in pictorial way



NEXUS in worst case makes twice optimizer calls from number of points lying on contour in high resolution discretized ESS. In worst case number of points lying on a contour can be D\*RES\*\*(D-1). So to construct M contours, worst case calls will be O(M\*D\*RES\*\*(D-1)).

This is again exponential in Dimension, and will incur more cost than Full ESS exploration if M\*D > RES. This is common, as too high resolutions are neither needed nor are computationally feasible for high dimensional ESS. During early development stages NEXUS is suggested to be a huge cost saving, which later turned out not to be that true, and for most tasks, Full ESS exploration turned out to be viable option.

From past works [Sanket, Sudarshan], It is clear that contours are either piecewise linear or can be approximated to be piecewise linear. See figure for reference of contours generated on a 50GB TPC-DS with Query instance Q91.



Our next improvement if to utilize this highly piecewise linear nature of contours to get improved and faster version of NEXUS, which should in practice speed up the contour discovery process, which is main cost overhead of compilation

We will use same seed discovery process of NEXUS, using binary search in interval of valid axis, from where we can start the search. Now consider that red contour (which is 4th contour in the diagram). Seed will be located on top boundary of ESS, and what if we could magically know the slope of contour in ESS space (do not consider cost into picture), which is nothing but infinity, as it is a straight line downward.

We could have used exponential search to reduce number of points on this contour from RES to log(RES). There are some fundamental issues with this approach:

1. Slope in ESS space for any piece of piecewise linear contour is not known a prior
2. Even if exact slope can be approximated somehow, Exponential search may miss some plans on contours due to large steps taken

First we will see how to overcome the second issue in our idea. We will be using bisection search to find if we can find a different plan in between two successive points discovered by Exponential search, and if a different plan if observed in either half, recursive function is called till any interval on bisection search has same plans on both end points, or interval length vanishes.

Also, Plan bouquet relies on low contour density, which is obtained at Intra-contour level using Anorexic reduction, once we have obtained all points on Contour. While Spill bound doesn’t rely due to contour density independent execution. We will be using a form of Anorexic reduction in bisection search described above to reduce the compilation efforts itself.

Now coming to the first issue, which is getting slope of each piece of piecewise linear function. We will try to solve this as online control system problem with feedback and fallback strategies.

As we know even with original NEXUS and Full ESS exploration, a tolerance interval of [C, (1+a)\*C] (with sufficient low value of a, say, 0.05) is used and points are chosen such that surface thickening is avoided, which is points must be chosen as close as possible to lower bound of search cost interval.

We will exploit same idea to search within a cost interval of [(1-e)\*C, (1+e)\*C], and our search method will try to pick point having optimal cost close to middle value of this interval.

With the knowledge of seed, we will start from one end of contour (one of the many connected linear pieces), and will search in 4th quadrant. Slope information will be obtained on the fly and tuned so that search will always lie within the given interval. Also, if exponential search goes beyond the interval, we always have a fallback to last valid point and start gain with half the step size taken in last wrong decision.

Also, a common observation is that, all slope changes of contour will be maximum of a right angle. In worst case, which is also observed in 5th contour in figure above, contour will take a sharp right angle turn anti-clockwise in 4th quadrant. No fallback in exponential search can get you the correct next direction. If we have detected that fallback and smaller steps will not work, we will go with exponential rotation to get correct next slope. This method of dynamic tuning of slope with exponential and bisection search will require far less optimizer call for piecewise linear contours, which are observed in practice.

**4.b Using Geometric progression on each axis of ESS with bounded MSO guarantee**

Uniform Selectivity distribution with high resolution is what Full space exploration and NEXUS needs in practice. But from past work on Concavity [SRINIVAS], we know that most of changes in cost value occur close to original. This can be captured in the following ways:

1. Sufficiently High resolution with uniform distribution
2. Selectivity values should be chosen carefully in geometric progression

While, for High dimensional queries, it is not possible to go with first choice of high resolution. In this sub-section we will work on usage of selectivity values on each axis from geometric distribution, and relaxed MSO bounds by going with this choice.

<PROOF FOR USAGE OF GEOMETRIC PROGRESSION>

**4.c Efficient computation of Inflated MSO guarantee**

With use of Improved NEXUS and ESS with Geometric distribution of selectivity values on each axis, we will obtain contours with far less number of points on each contour than the Full Space exploration and Uniform distribution of selectivity values.

In the existing region of ESS after database scale up. We’ll use these points with FPC module to get relaxed MSO guarantee if we will continue to use old contours and their plans.

At first we will look on computing nearly accurate value of guarantee values, under assumption of perfect model

While above method provides tight bounds on empirical MSO, it is Exponential in nature, and overall complexity will be O(M\*RED\*\*Dim). This will be overkill of resource, if volume of data changes frequently. Since we have lesser than exponential algorithms for compilation itself.

Now, we will look at an algorithm for efficient computation of MSO guarantee.

Max\_ratio = r\_pb;

For( ix\_ix=1 ; ic\_ix<m ; ic\_ix++ )

{

Early, Next = ic\_ix, ic\_ix+1

Min\_early = Min(FPC[plan\_id, Location] for (plan\_id, loc) in Contour[Early] )

Max\_next = Max(FPC[plan\_id, Location] for (plan\_id, loc) in Contour[Next] )

If ( Max\_next/Min\_early > Max\_ratio )

{

Max\_ratio = Max\_next/Min\_early ;

}

}

Compute MSO guarantee using Max\_ratio instead of r\_pb

Also, using similar algorithm as above, we can get information on which contour should be re-computed from scratch using NEXUS++, to lower down relaxed performance guarantee

Greedy choice of Contour to re-draw

<GREEDY CONTOUR CHOICE (GCC) ALGORITHM>

**4.d Incremental bouquet maintenance algorithms**

Layer wise development approach





**4. e Complete Pipeline for Incremental bouquet maintenance**

1. Check if any sub-ESS has any contour missing, if so draw missing contours in each ESS
2. Compute Weakened MSO guarantee using Min-Max algorithm, if they are tolerable halt incremental compilation prodedure
3. Select Contours in each sub-ESS to re-draw to improve performance guarantee
4. Run incremental bouquet algorithm for selected contours, use information from past contour
5. Repeat Step 2

**4.e Summary**

We have first reduced compilation overheads from what is done in past literature. Later we have shown efficient methods to determining relaxed guarantees from using a old and extended plan bouquet, and at last provide incremental bouquet maintenance algorithms

**5 EXPERIMENTS**

Hello

**6 CONCLUSIONS**

Done

**7 FUTURE WORK**

**7.a SELECTIVITY INDEPENDENCE**

**7.b DIMENSIONALITY REDUCTION**