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

MTech Project Report (CSA)

Achint Chaudhary

April 10th, 2020

**ABSTRACT**

OLAP applications often require a certain set of canned queries to be fired on database with varying the constants in query templates. 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.

**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 normalized in range of is known as 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 oracular optimal performance possible from all the available plan choices.

For each given query, predicates prone to selectivity error contribute as dimension in ESS is a multi-dimensional hypercube. The set of optimal plans over the entire range of selectivity values in ESS is called 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 . An is collection of all points from OCS which have same cost of optimal plan at each of these locations cost.

A subset of POSP is identified as , which is obtained by the intersection of Plans trajectories with OCS, creating multiple Iso-cost surfaces, each of which is placed at some cost-ratio () from the previous surface. Below figure 1 depicts and exemplar OCS and its intersection with IC trajectories for a 2-Dimensional ESS

A screenshot of a cell phone

Description automatically generated

Fig 1. OCS and Plan Trajectories intersection

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

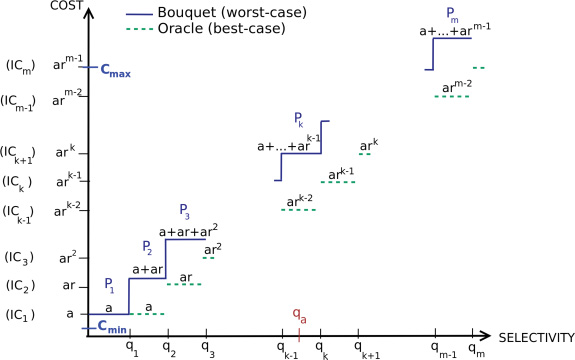


Fig 2. Cost incurred by Oracular vs Bouquet Execution

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 leads to Sub-Optimality (ratio of incurred cost to optimal cost) of bouquet approach as

This value is minimized using , 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 , where is maximum cardinality of plans on any of iso-cost surface.

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

Later an improved algorithm called [Cit:SRINIVAS] 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

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

**2. PROBLEM FORMULATION**

**2.a Notations**

|  |  |
| --- | --- |
| **Notation** | **Description** |
|  | Selectivity Predicates |
|  | Well Known Predicates |
|  | Error Prone Predicates |
|  | Trivial Predicates |
|  | EPP Selectivity Space |
|  | Optimal Cost Surface |
|  | Parametric Optimal Set of Plans |
|  | Resolution of Discretized ESS |
|  | Dimension of ESS |
|  | Cost of Optimal Plan at selectivity |
|  | Scaling Factor of Predicate |
|  | Minimum Selectivity on Predicate |
|  | Number of Iso-cost Contours |
|  | Iso-cost contour with index |
|  | Cost Budget of |
|  | Cost Ratio of Iso-cost contours |
|  | Selectivity interval for an axis of  Discretized ESS |
|  | Plan with assigned identity |
|  | Plan Cost Function for Plan |
|  | Cost of plan P at location q in ESS  of reference database |
|  | Cardinality of predicate at location which has undergone change of w.r.t to reference database |
|  | Difference in consecutive selectivity values on axis of ESS |
|  | Ratio of consecutive selectivity values on axis of ESS |
|  | Worst case slope of Plan Cost Function |
|  | Tolerance of contour thickening |

**2.b Assumptions**

**2.b.1 Plan Cost Monotonicity (PCM)**

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

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

Also, we assume that Plan Cost functions & OCS are continuous & smooth in nature

**2.b.2 Axis Parallel Concavity (APC)**

This assumption as stated in [Cit: SRINIVAS] is on Plan Cost Function () which is not just monotonic but exhibit a weak form of in their cost trajectories. For 1D , is said to be concave if for any two selectivities locations from and any below condition holds

Generalizing to D dimensions, a PCF is said to be if the function is concave along every axis-parallel 1D segment of .

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 exhibiting APC is that the corresponding , 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.

**2.b.3 Bounded Cost Growth (BCG)**

BCG property as defined by [Cit: DNC17], is as follows for a

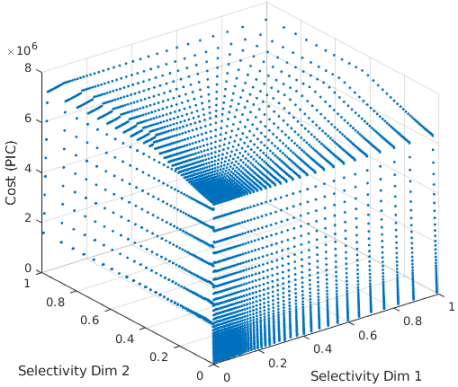
Here is an increasing function. Increase in selectivity by will result in maximum cost increase by factor of .

Similar to APC, BCG if holds for all PCF also holds for OCS.

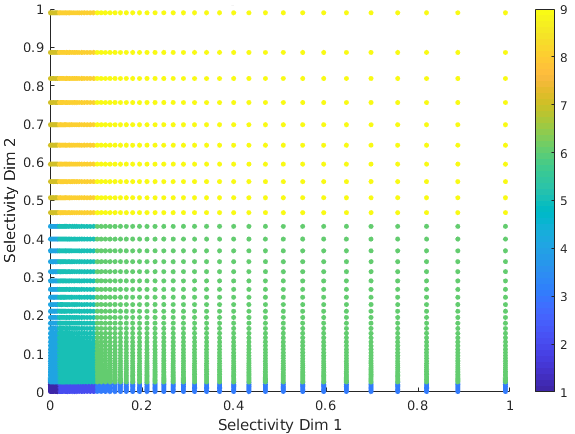
They have also claimed that identity function suffice in practice.

**2.b.4 Piecewise Axis Parallel Linear (APL)**

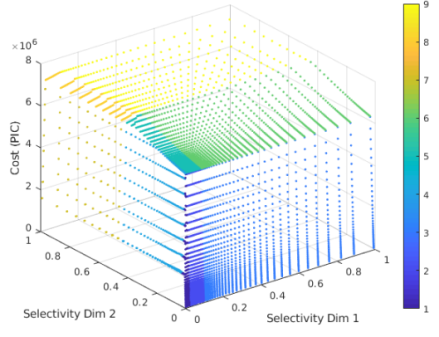
Plan Cost functions & OCS are shown to be piecewise linear in [Cit: HS02]. This property commonly comes from the fact that partial derivatives of common physical operators (except the sort operator, which is seldom found in industry strength benchmark [Cit: DNC17]) are linear in nature.



(a) Original OCS



(b) Partitioned OCS Domain



(c) OCS fitted with Piecewise functions

Fig 3. Multiple APL functions to fit OCS

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 [Cit: SANKET].

While in our work, we 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 which could itself be exponential in nature.

**2.b.5 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 WKP and TP, and for now we wish to handle all kind of predicates in same picture of analysis.

**2.b.6 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 [Cit: Wentao], proves most actual cost values to lie in 30% of estimated cost values after tuning.

**2.b.7 Selectivity Independence**

We assume that selectivity of predicates is 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**

**2.c.1 Sub-optimality**

It is ratio of cost incurred due to wrong selectivity estimation, as compared to optimal cost possible when actual selectivity is known a prior to system

Here,

This definition can be extended to plan bouquet where multiple executions takes place in a sequence with their respective budgets. So, definition will be

**2.c.2 Worst case Sub-optimality**

Worst case suboptimality is Sub-optimality w.r.t to that causes maximum Sub-optimality

**2.c.3 Maximum Sub-optimality (MSO)**

Global worst case is with all possible combinations of and over ESS, which is

MSO for a sequence of execution from bouquet will be

Theoretical guarantee on MSO is denoted as . While, empirical obtained MSO is denoted as

**2.d Updated notion of Selectivity Intervals**

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].

Now, we will look at type of changes in data stored in a database instance. It can be:

1. Distributional Change
2. Volumetric Change

Since plan bouquet and later devised techniques are robust to distributional changes. In the case when only data distribution has changed and all predicates are Error-prone, a plan bouquet compiled in past can be re-used with same

While, in the case of increase in size of database, Number of tuples to process and maximum possible tuples for any predicate generally increase.

Also, optimizer’s choice for physical operator is highly dependent on number of tuples.

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.

This difference of cardinality for same selectivity value in both instances will led to change in choice of operators and may result into overall change in structure of plans. This will ultimately change both shape of iso-cost contours as well as plans lying on them.

When we are looking to use information from contours generated on earlier instance for incremental or faster compilation. This uniform picture of [0,1] selectivity will bring problem in easy analysis.

So, we will chose to keep uniform selectivity to cardinality mapping across multiple sized database instances.

In a loose sense, selectivity to cardinality mapping should not be changed across different sized instances, while we opt to change the selectivity interval for our analysis.

For example, if total tuples in a relation before update are 100, and post update it has become 200. Then, we wish that 0.5 selectivity on both databases should return 50 tuples (this 50 is with reference to earlier instance). So for initial database selectivity values are from [0,1], while for instance post update, legal selectivity interval is [0,2].

Note that, only the base (reference) instance on which plan bouquet is first compiled has selectivity values in [0,1]

This may seem to be fuzzy at first, but we will later show importance of this notation for reducing cost of incremental compilation.

Also, we will look change in volume from ration of change in maximum cardinality possible from a predicate. This comes from that notion that each axis of ESS denotes a predicate. Hence, ESS upon change in database may have grown differently on each axis of ESS.

A pictorial representation of this approach is shown. Also, potential regions for which Incremental algorithms are needed to be developed are mentioned in the fig 4.





Fig 4. Representation of Different regions of concern in cardinality space, with new approach of selectivity intervals

In above diagram. Green region is the portion for which standard compilation procedure of contour discovery need to be called.

For now, we will look for approaches to

1. Extend contours which lie at intersection boundaries of red & blue regions into blue region from red region.
2. Check impact of update in cost of points lying on contours with information of their location in ESS and optimal plan at each individual point. This calculation will be helpful to provide relaxed from using old contours and plans within red region
3. Incrementally compile contours in red regions. If usage of any old contour is deteriorating performance guarantee significantly

**3 CHALLENGES W.R.T UPDATES**

Under situation of change in database instance, placements of ideal contours can be totally different from existing contours built from earlier database instance, which may result that old bouquet contours and plans on them are totally different from new bouquet contours. Under above mentioned conditions it seems that a re-compilation will be needed.

Plan bouquet is suitable for canned queries as compilation overhead of entire discretized ESS enumeration will take cost, which is amortized over repeated invocations for canned queries.

Now we will look at how compilation (iso-cost surface identification) in past literature is done [Cit: Rajmohan, Dutt], and why under all present options compilation at it first place is resource exhaustive.

**3.a Math behind compilation**

& are the extreme ends of ESS, with having minimum possible and maximum possible selectivity of each error-prone predicate respectively. Cost of optimal cost at both these points obtained via Selectivity Injection are denoted as and respectively. using these two values, number of iso-cost contours () is obtained as follows.

These Iso-cost contours are drawn at cost ratio successively from which is referenced as , as first term of cost geometric progression. Last contour may be at ratio less than from .

Let’s have a proof by cases that this will not impact .

Case 1: Actual selectivity is discoverable up to execution of plan from or any contour before than that, let that contour be . In that case for 1D Plan bouquet

Case 2: Actual selectivity is discovered on execution of plan from . In that case

Let this ratio of from be . Also, . Now using expression for sum of geometric progression, which is used evaluate for plan bouquet is

To maximize this upper bound, we substitute upper bound of which is . Hence, resulting expression will be

In both Case 1. and Case 2. We have got same final expression, while in case 2. We have used upper bound for . This mean expression from Case 1 provides .

When is substituted in final expression, sub-optimality in that case is also upper bounded by 4.

This same proof can be easily extended for multi-dimensional plan bouquet.

Hence, we can state that last contour can be placed at lesser than cost ratio, yet will provide same .

**3.b Compilation Methods & Overheads**

Next step of compilation is to identify selectivity location and their optimal plans for each of iso-cost contours.

For now, there are two options available for contour construction:

1. Full ESS enumeration
2. NEXUS.

Let’s see them one by one

**3.b.1 Full discretized ESS Enumeration**

This is most naïve yet effective approach and will be referred as full space enumeration at most places. In this approach 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 optimizer calls. Where is resolution chosen to discretize ESS, while is dimension of ESS. Each dimension in ESS represents a error-prone predicate.

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

**3.b.2 NEXUS (NEighborhood Exploration Using Seed)**

An optimization over full space enumeration is introduced with plan bouquet [Cit Rajmohan Dutt]. NEXUS is an algorithm proposed to avoid making unnecessary optimizer calls on points lying in between contours. If we have total iso-cost contours to discover, worst case complexity of NEXUS for entire compilation process can go up to

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 predicate’s selectivity through domain knowledge, Spillbound can shrink ESS by making this lower bound as origin. However NEXUS needs to redraw new iso-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 enumeration

NEXUS is worst case makes total optimizer calls twice the number of points lying on iso-cost contours

**Note:** 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 with cost of optimal plans at these locations lying in interval even with small values of , say, 0.05.

Due to this assumption, we will see some issues which are common to both Full ESS enumeration and NEXUS

**3.c Complexity Issues in compilation**

Both methods have a common property with them:

1. Complexity exponential in
2. Need of sufficiently high resolution on each axis

While an algorithm with complexity is expensive and computationally unfeasible to run with sufficiently high resolution, specially in high dimensional ESS.

To prevent this in practice, instead of going with sufficiently high resolution with uniform distribution of selectivity values on each axis. Experiments could be tried to run on low-resolution picture.

Next we will see a potential issue possible with both current methods with use of low resolution with uniformly distributed selectivity values.

But before that we will discuss both methods in brief for a 2D ESS example, to find points lying on contour with cost

1. **Full space enumeration**

Points lying from grid in cost interval should be considered to be on contour if cost of optimal plan lies in

1. NEXUS

Locate seed , then iterate untill loop ends

Due to usage of low resolution, with low tolerance factor . Full Space enumeration may result in to incomplete contour, while NEXUS may result into contour will cost inflated more than factor of . Pictorial representation of potential issues encountered are depicted in figure 5.

A screenshot of a cell phone

Description automatically generated

Fig 5. Contour discovery with low resolution and uniform selectivity distribution

To avoid this yet keeping computational feasibility, one possible option is to raise value of . But that does change abruptly and from observation and also from APC, we know most changes in slope will happen close to origin.

So, to avoid (empirically) above explained issue, when working with low resolution and high dimensional ESS and also to keep cost 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 in practice.

This violation is not observed till yet, but proof for the same is also pending like a . Rational till now for use of Geometric distribution in selectivity space is that it captures many points in low selectivity values, and most changes in plan choices takes place in low selectivity values.

For making a geometric distribution to work, there are numerous hyper-parameters to tune. While methods, tips and techniques along with their impact on are the missing part from literature which we will try to provide and prove in a systematic way.

**4. CONTRIBUTIONS**

**4.a Increasing compilation efficiency**

Few comments on NEXUS as discussed in [Cit: SRINIVAS THESIS] highlights some issues faced due to use of NEXUS. Two such major issues 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.

Fraction of speed-up of NEXUS over Full space enumeration is

As number of dimensions and number of contours to draw m increase, both are cases with queries with more number of . Also, to make things feasible we choose low values of . All this aspect of moving towards tractable compilation time brings NEXUS close to Full space enumeration.

We will propose a upgrade in NEXUS to make it much more faster than its competitor naïve algorithm.

But before we start working on a improved version of NEXUS, we will first try to look at second argument made against NEXUS

**4.a.1 Impact of known lower bound**

Consider a 1d example of plan bouquet. If lower bound on predicate selectivity known a prior is . Then selectivity interval will reduce from to .

Let selectivity locations for each of contour be . So,

Here,

Let, lie beyond , and actual selectivity be discover upon execution of

So, contours through are no longer needed. Expression for suboptimality will be changed from

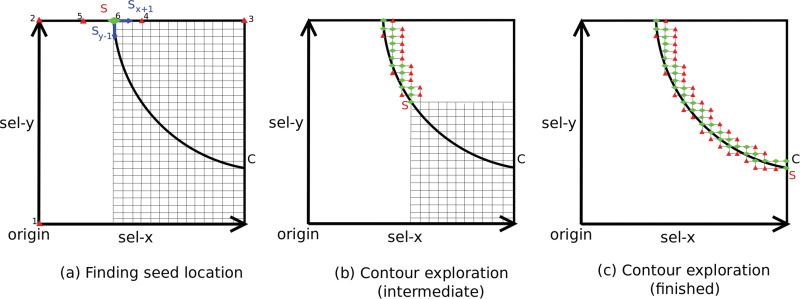
To

This is same expression for as seen in earlier contours. So, we can now state that with knowledge of lower bound of a predicate’s selectivity, we can discard contours having selectivity points having selectivity of that predicate less than lower bound. But still will receive same

Hence, now we are clear that knowledge of lower bound neither impact NEXUS nor Full Space enumeration, contours drawn in past can be continued to use with any change in knowledge from lower bound.

We will revisit NEXUS and see scope of improvement based on some geometric properties and try to put on improvement into existing NEXUS algorithm.

Base idea of NEXUS is first locate a seed, which is one end point of contour; Use this seed to discover adjacent points of contour lying in 4th quadrant in its neighborhood. Below example borrowed from [Cit ANSHUMAN PAPER] shows working of NEXUS in pictorial way



A picture containing game

Description automatically generated

A picture containing game

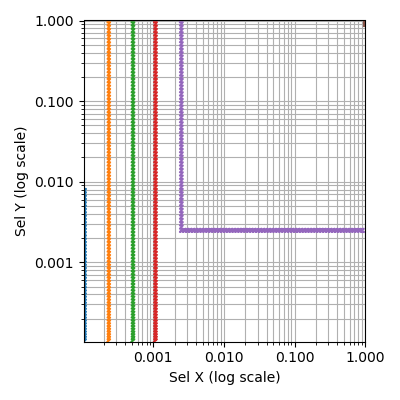
Description automatically generated

Fig 6. Working of NEXUS

NEXUS in worst case makes twice optimizer calls from number of points lying on contour in high resolution discretized ESS.

This at first glance looks bit smart, but we can still improve it using piecewise linear geometry of contours.

From past works [Cit: Sanket, Sudarshan], 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.



We’ll attempt to utilize this highly piecewise linear nature of contours to get improved and faster version of NEXUS, namely 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 as of NEXUS, using binary search in interval of valid axis, from where we can start the search. Next we’ll look at design of contour exploration of NEXUS++

As an example, consider that red contour (which is 4th contour in the diagram). Seed as usual will be located on top boundary of ESS.

Now, 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 contour is parallel to Y-axis.

We could have used to reduce number of points on this where optimizer calls are made. In best case complexity will change from to .

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 Geometric progression to discretize each axis of ESS with bounded**

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**

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** 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**

**8 REFERENCES**