Turbo-charging Plan Bouquet Identification

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Engineering
IN
Faculty of Engineering

BY

C Rajmohan

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2015

Declaration of Originality

I, **C Rajmohan**, with SR No. **04-04-00-10-41-13-1-10164** hereby declare that the material presented in the thesis titled

Turbo-charging Plan Bouquet Identification

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2013-15**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

My Family
for their love and support

Acknowledgements

I am deeply grateful to Prof. Jayant R. Haritsa for his unmatched guidance, enthusiasm and supervision. He has always been a source of inspiration for me. I have been extremely lucky to work with him.

I am thankful to Anshuman Dutt for his assistance and guidance. It had been a great experience to work with him. My sincere thanks goes to my fellow lab mates as well for all the help and suggestions. Also I thank my CSA friends who made my stay at IISc pleasant, and for all the fun we had together.

Finally, I am indebted with gratitude to my parents and sister for their love and inspiration that no amount of thanks can suffice. This project would not have been possible without their constant support and motivation.

Abstract

In database systems, the selectivities estimated for query predicates during the optimization phase often differ significantly from the actual selectivities encountered during query execution, leading to sub-optimal performance. Plan Bouquet is a recently proposed technique that eschews compile-time selectivity estimation and instead incrementally discovers selectivities through a sequence of partial executions from a bouquet of plans identified at compile time. Although this technique promises sub-optimality guarantees, its practicality is hindered by the overheads of identifying the bouquet of plans which requires (a) optimizing the entire selectivity error space; (b) reducing the bouquet cardinality to achieve practically useful guarantees.

In this work, we introduce a novel algorithm called *NEXUS* to identify the bouquet of plans efficiently without enumerating the entire selectivity error space. Further, we propose efficient variants of the existing bouquet cardinality reduction technique and show that the performance is comparable to the best reduction that can be achieved, even with limited knowledge about the selectivity space. The proposed techniques have been integrated within the prototype implementation of Plan Bouquet. Also the *ForcePlan* feature, necessary for bouquet cardinality reduction, has been implemented in PostgreSQL 9.4 database engine. With this setup, the evaluation of the proposed algorithms over TPC-H and TPC-DS benchmark queries demonstrates that (a) the preprocessing time reduces from *hours to minutes*, and (b) the memory footprint reduces from *GBs to MBs*, without affecting the performance guarantees of Plan Bouquet approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We interact with database systems through declarative SQL queries. Given an SQL query, there are typically a large number of alternative ways in which a database system could execute the query. Though it does not change the results, each alternative way of executing a given query varies widely in performance. Database systems have complex, cost based query optimizers which devise an execution plan for each query that they receive. Execution plan is an ordered sequence of steps to perform the operations requested by SQL query. The optimizer will generate and evaluate many plans and choose the least cost plan as best plan. The estimated cost of a plan is primarily a function of selectivity which is the estimated number of rows of each relation relevant to producing the final result. Optimizer estimates many selectivities while identifying the ideal execution plan for a query. In practice, these estimates are often significantly different with respect to actual selectivities due to a variety of reason[6], resulting in sub-optimal execution performance.

1.1 Background

The proposed techniques to solve the selectivity estimation problem (see [7] for a comprehensive survey) include improving the statistical quality of the meta-data, re-optimization techniques etc. each of which have their own limitations.

Recently, a different approach is proposed for handling error-prone selectivities, called Plan Bouquet[1] approach. It completely skips error-prone selectivity estimation during planning phase, rather query is executed through a calibrated sequence of cost bounded executions of a bouquet of plans. Plan Bouquet approach provides guaranteed upper bounds on worst-case sub-optimality for the first time in the literature but it requires a significant amount of preprocessing to be done to identify the bouquet of plans at compile-time.

Let us call the space formed by erroneous selectivities as Error-prone Selectivity Space (ESS). Firstly, the entire selectivity error space is optimized and the set of plans that cover the entire selectivity range are identified. These set of plans are called as *Parametric Optimal Set of Plans(POSP)*. Secondly, by locating the isocost contours on the ESS that are in geometric progression, a small bouquet of plans is identified from the POSP set such that at least one among this subset is near-optimal at each location in the space. In order to have practically useful guarantees, the POSP set is reduced using Cost Greedy FPC[3, 4] plan reduction algorithm before locating isocost contours. For this purpose, database engine should support FPC(Foreign Plan Costing) feature to cost plans outside their optimality regions.

1.2 Motivation

Producing the POSP set for the entire ESS requires repeated invocations of the query optimizer at a high degree of resolution over ESS. If the resolution of a d-dimensional ESS is 'res' and the number of plans on the ESS is 'n' then, the number of optimization calls made for producing the POSP is res^d, which is computationally expensive. In order to have practically useful guarantees, the POSP set is reduced using Cost Greedy FPC plan reduction algorithm before identifying isocost contours on the ESS. This requires $n.res^d$ number of FPC calls to be made, which is computationally expensive as well.

Optimized locations in the ESS need to be held in memory until anorexic reduction is performed leading to high memory overheads as well. The overheads increase exponentially with ESS dimensionality. This necessitates an efficient and scalable technique to be proposed that reduces preprocessing overhead of Plan Bouquet approach. Here, the challenge is to not only minimize the number of optimizer invocations and memory overhead but also ensure that the performance guarantees assured by Plan Bouquet technique are not affected.

Finally, anorexic plan reduction techniques used in bouquet identification phase require costing plans outside their native optimality regions in ESS motivating us to develop a generic *ForcePlan* feature which can also be used in bouquet execution.

1.3 Contributions

In this work, we propose techniques to reduce the preprocessing overhead of Plan Bouquet approach by identifying bouquet plans efficiently. Our contributions can be divided into 4 categories broadly.

- **Efficient Contour Identification:** We propose an efficient algorithm to identify the isocost contours directly without exploring locations that lie between the contours in the

ESS. We discuss the efficiency of our algorithm in terms of time and space.

- **Contour-centric Reduction of Plans:** We propose variants of the existing plan reduction techniques to reduce plans on the contours efficiently. We show that the performance of our proposed variants is comparable to the 'oracle' technique which knows all plans in the ESS and achieves the best possible reduction always.

- **ForcePlan Feature:** We describe our implementation of *ForcePlan* feature in PostgreSQL 9.4[10] database engine, which is required during plan reduction to cost plans at their exo-optimal regions. This feature guides the query optimizer into picking user specified execution plan as the plan of choice for a given query, bypassing the query optimization.

- **System Implementation:** We describe how the proposed bouquet identification techniques are integrated with Plan Bouquet prototype system and evaluated on PostgreSQL 9.4 database engine over TPC-H and TPC-DS benchmark environments. We discuss the modifications done in database engine and Plan Bouquet prototype system to migrate the system to the latest PostgreSQL engine. We show empirically that we reduce bouquet identification overheads to a great extent while preserving the performance guarantees of Plan Bouquet approach.

1.4 Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides the background details about Plan Bouquet technique. Chapter 3 formulates the bouquet identification problem. Chapter 4 and Chapter 5 elaborate contributions of our work in contour identification and contour-centric reduction of plans respectively followed by a detailed explanation of *ForcePlan* feature in Chapter 6. Chapter 7 covers integration of our work with Plan Bouquet prototype system. Experimental evaluation of our work is presented in Chapter 8. Finally, in Chapter 9, our work is summarized and future work is outlined.

Chapter 2

Plan Bouquet technique

In this chapter, we present necessary background details about Plan Bouquet approach for robust query processing, as given in Plan Bouquet[1] paper.

2.1 Overview

Plan Bouquet is a new approach wherein the compile-time estimation process is completely shunned for error-prone selectivities. Instead, these selectivities are learnt systematically at run-time by executing a carefully chosen small set of plans called "Bouquet plans" in a particular sequence in cost-limited manner. Partial executions are controlled by a graded progression of isocost surfaces projected onto the POSP curve. It has been proved in [1] that this construction results in guaranteed worst-case performance. It assumes Plan Cost Monotonicity (PCM) property, which states that cost of POSP plans increases monotonically with increasing selectivity values.

2.2 Single Dimension Example

Consider the simple query shown in Figure 2.1. This example query contains one base predicate selectivity and two join selectivities. Out of these, assume selectivity of base predicate $p\_retailprice < 1000$ is error-prone.

```
select * from lineitem, orders, part
where p_partkey = l_partkey and l_orderkey =
        o_orderkey and p_retailprice < 1000
```

Figure 2.1: Example Query (EQ)

2.2.1 Bouquet Identification

Firstly, the Parametric Optimal Set of Plans (POSP) that cover the entire selectivity range of the error-prone predicate is identified through repeated invocations of the optimizer and explicit injection of selectivities from lowest to highest values.

Let POSP set be comprised of plans P1 through P5. Further, each plan is optimal over a certain selectivity range. The costs of these five plans, P1 through P5, over the entire selectivity range are enumerated as shown in Figure 2.2(on a log-log scale), using foreign plan costing. From these plots, the trajectory of the minimum cost from among the POSP plans is obtained as a curve which represents the ideal performance. This curve is called as POSP Infimum Curve (PIC).

Next, the PIC is discretized by projecting a graded progression of isocost (IC) steps onto the curve. In Figure 2.2, the dotted horizontal lines represent a geometric progression of isocost steps, IC1 through IC7, with each step being double the preceding value. The intersection of each isocost step with the PIC (indicated by ■) gives an associated selectivity, and the identity of the best POSP plan for this selectivity.

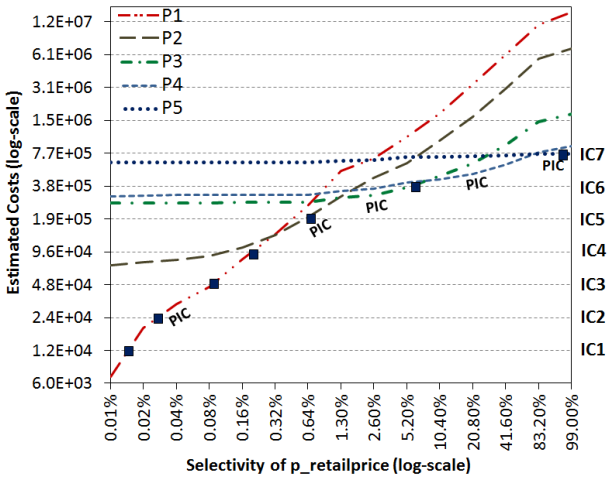


Figure 2.2: POSP performance

For example, in Figure 5, the intersection of IC5 with the PIC corresponds to a selectivity of 0.65% with associated POSP plan P2. The subset of POSP plans that are associated with the intersections forms the plan bouquet for the given query. So in Figure 2.2, the bouquet plans are P1, P2, P3 and P5.

2.2.2 Bouquet Execution

At run-time, through a sequence of cost-limited executions of bouquet plans, the actual query selectivities are discovered. Specifically, execution begins with the cheapest isocost step, and iteratively executes the bouquet plan assigned to each step with a cost budget. At each step, either

1. The plan does not complete execution within the allotted budget, which means the actual selectivity location lies beyond the current step, causing execution to be switched to next isocost step in the sequence; or

2. The current plan completes execution within the budget, which means the actual selectivity location has been reached, and a plan that is at least 2-competitive with respect to the ideal choice was used for the final execution.

2.3 Extension to multiple dimensions

In a multi-dimensional selectivity environment, the IC steps and the PIC curve become surfaces, and their intersections represent selectivity surfaces on which many bouquet plans may be present.

For example, in the 2-D case, the IC steps are horizontal planes cutting through a hollow 3D PIC surface, typically resulting in hyperbolic intersection contours with different plans associated with disjoint segments of this contour, an instance of this scenario is shown in Figure 2.3.

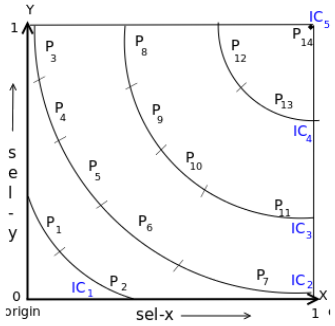


Figure 2.3: 2-D Selectivity Space

2.4 Robustness Metric

Robustness is measured in terms of maximum sub-optimality **(MSO)** for plan bouquet. Given a query Q and an ESS, let q_e denote the optimizer's estimated query location, and q_a denote

the actual run-time location in ESS. Also, denote the plan chosen by the optimizer at q_e by P_{oe} and the optimal plan at q_a by P_{oa}. Finally, let cost(P_j, q_i) represent the execution cost incurred at actual location q_i by plan P_j. Then, robustness is defined by the normalized metric:

$$MSO = \max_{q_e, q_a \in ESS} [\frac{cost(P_{oe}, q_a)}{cost(P_{oa}, q_a)}]$$

which ranges over [1,∞). Plan Bouquet gives robustness guarantee in terms of MSO as:

$$MSO \leqslant 4\rho$$

If there exist K total contours and n_i plans lie on i^{th} contour then,

$$\rho = \max_{i=1\ to\ K} n_i$$

It is clear from the above expression that bouquet MSO is a direct function of POSP plans, so MSO can be decreased by reducing number of plans in POSP. It can be done through anorexic plan reduction which decreases number of plans in POSP without substantially affecting the quality of query processing for any individual query in the selectivity space.

2.5 Plan Bouquet prototype system

QUEST [2](QUery Execution without Selectivity esTimation) is a prototype implementation of Plan Bouquet approach. It visually shows bouquet execution process. Picasso [5] tool is used in preprocessing phase to identify POSP set and then reduce the POSP set in ESS. The preprocessed data is then used by QUEST to pick bouquet plans on the contours and execute them in a carefully chosen manner as explained in Section 2.2.2. Currently, PostgreSQL 8.3 [9] database engine is enhanced with required features and used by QUEST for bouquet execution.

Chapter 3

Problem Formulation

3.1 Overheads in Bouquet Identification

Existing Bouquet Identification approach[2] follows a brute-force mechanism to identify contours, wherein the entire ESS is optimized first, then anorexic reduction is performed on the entire ESS using Cost Greedy FPC[3, 4] technique to get reduced plans. Finally contours are identified by searching through the entire ESS. This approach mainly suffers from the following overheads.

Time Overhead: While optimizing the ESS, in order to determine the cost of any ESS location we need to invoke the database query optimizer to perform optimization which is a time-consuming operation. Though it is negligible for a single optimization call, it is a very high computational overhead when it comes to finding bouquet plans in higher dimensional ESS since millions of optimization calls will be made.

Space Overhead: Plan Bouquet MSO is dependent on number of plans on the densest contour. As contour plan density is typically huge for higher dimensions, plan reduction techniques are used to reduce it. For this purpose, all optimized locations are kept in memory till anorexic reduction. Due to this, memory space becomes a bottleneck especially in higher dimensions.

To understand the amount of overhead in preprocessing, consider an example query having 5 dimensions in ESS with a resolution of 50. Then approximately, it takes around 30 days to optimize the entire ESS (with PostgreSQL 9.4 database engine) and 4.7GB of memory to maintain the ESS locations.

3.2 Problem Statement

Given a d-dimensional ESS space, we want to find bouquet plans in a time-efficient manner with minimum invocations to optimizer without affecting the performance guarantees of Plan Bouquet technique. We need to ensure that only the required ESS locations are maintained in memory during preprocessing to avoid memory overhead in higher dimensions.

Formally, given a set of contour costs C = $\{C_1, C_2, ...C_n\}$ such that C_i's are in geometric progression, *Bouquet Identification* needs to find a set P = $\{P_1, P_2, ...P_n\}$ where P_i is a minimal set of plans such that there exists atleast one plan in the set P_i that is optimal for each location of the ESS where cost is C_i.

Since Bouquet MSO depends on contour plan density and plan density is typically high for a higher dimensional ESS, we split this problem into two sub-problems namely *Contour Identification* and *Contour Plans Reduction* as follows.

Contour Identification: Given an ESS and a set of contour costs C = $\{C_1, C_2, ...C_n\}$, *'Contour Identification problem'* is to find all locations in ESS having cost equal to C_i for each i efficiently.

Contour Plans Reduction: Given a set of isocost contours IC_i, $i = 1 : n$ with P_i plans on contour i, *'Contour Plans Reduction problem'* is to find a reduced plan set R_i with minimum plan cardinality for each contour i where $R_i << P_i$(given P_i is not small) such that cost of no isocost contour location in ESS increases by more than λ%(near optimal).

The bouquet identification techniques should not affect performance guarantees assured by Plan Bouquet approach. Further, Cost Greedy FPC plan reduction requires costing plans at their exo-optimal regions in ESS, which compels *ForcePlan* feature in database engine. Specifically, given a query Q and a feasible plan P, we need to force the optimizer to use plan P as the plan of choice bypassing the default optimization process.

Chapter 4

Efficient Contour Identification

Consider an ESS grid with a fixed resolution 'res' on each dimension. An isocost contour of cost C is a contour formed by locations in ESS having cost = C. It is to be noted that the discretized ESS may not contain locations with optimal cost exactly C. In such cases, it is safe to accept a neighbouring location q with $C < c_{opt}(q) \leq (1+\epsilon)C$ as a substitute, where $c_{opt}(q)$ is the optimal cost of query location q and ϵ is a tolerance factor. We assume that the ESS grid resolution is sufficiently high such that we can always find adjacent approximate locations with small value of ϵ, say 0.05.

Acceptable contour in a grid: Let us define the notion of an acceptable contour with a 2D ESS grid. Say for a given location L with coordinates (x, y), the location $(x+1, y)$ is denoted with L_{x+1}, location $(x-1, y)$ with L_{x-1} and location $(x-1, y-1)$ with L_{-1} (L_{y+1} and L_{y-1} are also similarly defined).

Let us say that we want to find the contour of cost C. In this setting, location $L_{(x,y)}$ is acceptable for contour cost C if it satisfies following conditions,

- [*Validity check*] $C \leq c_{opt}(L) \leq (1+\epsilon)C$
- [*Non-redundancy check*] If $c_{opt}(L_{x-1}) > C$ and $c_{opt}(L_{y-1}) > C$ then $c_{opt}(L_{-1}) < C$

Here the first condition ensures the validity of a contour location by stating that the optimal cost of the location must be equal to C or above C but within a tolerance factor of cost C. The second condition ensures that we do not consider any location which is redundant.

Figure 4.1 provides visual explanation for the acceptable contour. Here, say all marked(o) locations in the ESS are valid. Out of these locations, the acceptable contour locations are shown in green colour and redundant locations are shown in blue colour. For example, location L is redundant since location L_{-1}(which is valid) itself covers the third quadrant of the ESS

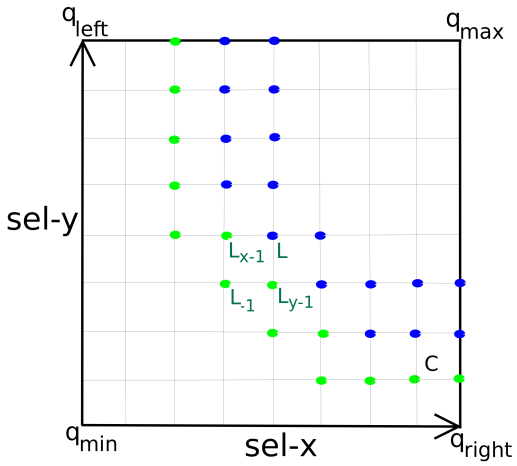


Figure 4.1: Acceptable contour in 2D ESS

region that can be covered by L in all dimensions as far as the current contour is concerned. Note that if we had taken the blue locations rather than green locations as acceptable contour locations, then the space covered by the contour would have been more. But, this requires tuning parameter ϵ according to the context which is unnecessary. Hence, we stick to green locations as acceptable contour locations.

Our Approach: The isocost contours can be identified through a brute force exploration of the ESS. However, the overheads of such exploration increase exponentially with ESS dimensionality. But, in principle, the bouquet execution can be started with plans from only the first contour and plans on other contours can be identified in parallel with the partial execution of plans on earlier contours.

With the above observation, we present an approach that can quickly identify query locations for a particular isocost contour without exploring locations that either correspond to other contours or lie between the contours in the ESS. We exploit the basic assumptions of monotonicity and smoothness of plan costs which together imply that the optimal costs are in ascending order with respect to each dimension in the discretized ESS and do not change abruptly. This also means that the problem is similar to searching for an element in a sorted matrix with n^2 elements, which can be solved in $O(n)$ using Saddleback Search[8].

Firstly, we present the algorithm for 2D ESS, that explores the ESS in order to find locations with a given optimal cost C, followed by its extension to higher-dimensional ESS. Our goal is to reduce the number of optimization calls and the number of optimized locations to be maintained in memory while identifying the contours, to achieve efficiency in time and space respectively.

It is to be noted that identification of contours is simple in one dimensional case since any

contour corresponds to exactly one location (and hence one plan as explained in Chapter 2) which can be located using binary search over the ESS locations that are sorted on cost due to PCM property.

4.1 Algorithm for 2D ESS

Assume that the 2D ESS is available in the form of a grid ($res \times res$) bounded by $q_{min} = (1, 1)$ and $q_{max} = (res, res)$ as shown in Figure 4.1, where 'res' is grid resolution. Then, any valid isocost contour on the ESS must have cost C such that, $c_{opt}(q_{min}) \leq C \leq c_{opt}(q_{max})$ and satisfy the following properties,

Property 1: For an isocost contour of cost C, there must be a location q_s on the left/top boundary of the ESS.
Proof: It is given that,

$$c_{opt}(q_{min}) \leq C \leq c_{opt}(q_{max})$$

Further, PCM implies that,

$$c_{opt}(q_{min}) \leq c_{opt}(q_{left}) \leq c_{opt}(q_{max})$$

Hence, there are two cases: (a) $c_{opt}(q_{min}) \leq C \leq c_{opt}(q_{left})$ or (b) $c_{opt}(q_{left}) < C \leq c_{opt}(q_{max})$. In case (a), there must be a location q with cost C on the left boundary due to the assumptions of smoothness of plan costs and high grid resolution. Similarly, in case (b), there must be a location q with cost C on the top boundary.

Property 2: For an isocost contour of cost C, there must be a location q_f on the right/bottom boundary of the ESS.
Proof: Similar to the proof of Property 1.

Property 3: For a given isocost contour, the location q_f must lie in the 4^{th} quadrant of q_s.
Proof: This property is true because for a given q_s, either 1^{st} and 4^{th} quadrant lie in ESS (q_s on left boundary) or 3^{rd} and 4^{th} quadrant lie in ESS (q_s on top boundary) but q_f cannot lie in 1^{st} and 3^{rd} quadrant of q_s due to PCM.

With the above, to identify a contour of cost C in ESS, our idea is to locate q_s as initial seed location on the contour, then grow the seed along its neighbourhood in small steps to trace the complete contour until it reaches q_f. Thus, the algorithm works in two phases:

1. Locating the initial seed
2. Neighbourhood exploration using seed

4.1.1 Locating the initial Seed

Let us locate the seed that has maximum 'y' coordinate which corresponds to the location q_s. There are two possibilities (a) y < res, (b) y = res.

To this end, we need to explore only the left and top boundaries of the ESS. Firstly, the correct boundary is determined by costing the end points of the boundaries and secondly the initial seed location (S) is determined by binary search along the boundary.

4.1.2 Neighbourhood EXploration Using Seed (NEXUS)

Here we trace the contour curve by growing the initial seed found in last step along its neighbourhood. The primary observation here is that, with maximum 'y' location as origin, only 3^{rd} and 4^{th} quadrants are present and the locations in the 3^{rd} quadrant are already known to be unacceptable due to PCM. With the above observation, the initial seed location S can be used to recursively generate new seed locations in the 4^{th} quadrant and thus grow the isocost contour.

For a given seed location $S_{(x,y)}$, having $C \leq c_{opt}(S) \leq (1+\epsilon)C$, we find optimal cost for new candidates for seed, i.e., S_{x+1} and S_{y-1}. Here, PCM implies that $c_{opt}(S_{x+1}) > c_{opt}(S)$ and $c_{opt}(S_{y-1}) < c_{opt}(S)$. Now, the new seed is chosen using the following criterion,

- If $c_{opt}(S_{y-1}) \geq C$, then set $S = S_{y-1}$
- If $c_{opt}(S_{y-1}) < C$, then set $S = S_{x+1}$

If neither of S_{x+1} or S_{y-1} exist in the grid, the algorithm ends since it implies that the contour has been completely explored.

Figure 4.2 provides visual explanation for the above algorithm. Figure 4.2(a) shows the process of identifying the initial seed location(S) using 6 optimization calls. As the contour exploration proceeds in the 4^{th} quadrant of S, as shown in Figure 4.2(b), the optimized locations are marked with a symbol (red colored Δ or green colored o) - those marked with green constitute the accepted contour locations and red marks the locations that were explored but rejected. Figure 4.2(c) shows that the contour exploration is complete when S hits the ESS boundary and hence no more seeds could be generated. Here, it is worth noting that red Δ = green o leading to the following result.

Theorem 1: NEXUS algorithm performs exactly *twice the number of optimization calls* as compared to the optimal algorithm that finds only acceptable contour locations.

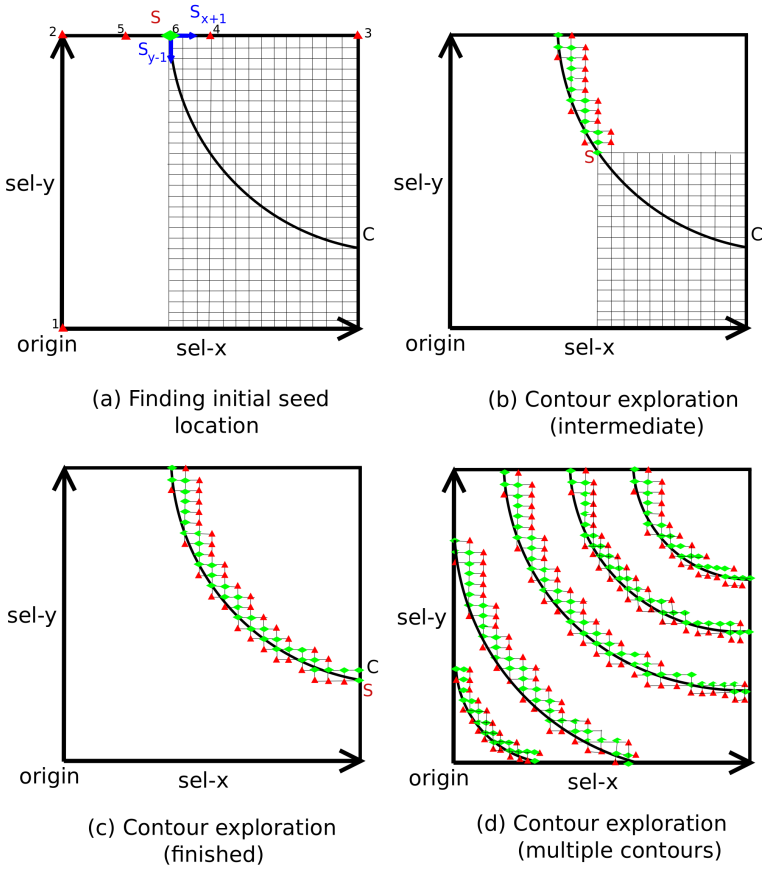


Figure 4.2: Contour Identification in 2D ESS

Proof: During contour exploration by NEXUS algorithm, at any location S there are exactly two candidates for new seed(S_{x+1} and S_{y-1}) and one of them is definitely on the (accepted) contour. Hence NEXUS algorithm optimizes exactly twice the number of locations(ignoring the small number of optimization calls made for locating the initial seed) as compared to the optimal algorithm that optimizes only acceptable contour locations.

Also note that when the contour exploration is at seed location S, if location S_{y-1} is found to be on the (accepted) contour after optimization, then there is no need to optimize S_{x+1} location since the new seed will be S_{y-1}.

A similar explanation can be given for tracing an isocost contour that starts at left boundary rather than top boundary using the NEXUS algorithm.

The pseudocode for identifying a contour is given in Algorithm 1. This algorithm takes two arguments – *ESS* is the error-selectivity space with resolution *res* and *contourCost* is the cost of the contour to be identified. The following are the external functions referred in the pseudocode.

Algorithm 1 2D Contour Identification Algorithm

```
FindContour2D(ESS, contourCost)
/* Locate the initial seed */
topleft_point_cost = ESSPointCost(ESS, 1, res)
if topleft_point_cost > contourCost then
    /* binary search along left most edge */
    x = 1
    y = BSearchEdge(1, contourCost)
else
    /* binary search along top most edge */
    y = res
    x = BSearchEdge(0, contourCost)
end if

/* Initial seed S_(x,y) */
AddContourLocation(ESS, x, y, contourCost);

/* Trace contour curve by neighbourhood exploration using seed */
while x < res and y > 1 do
    cand1Cost = ESSPointCost(ESS, x + 1, y)
    cand2Cost = ESSPointCost(ESS, x, y − 1)
    if cand1Cost ≥ contourCost and cand2Cost < contourCost then
        /* New seed S = S_{x+1} */
        x = x + 1
    else if cand1Cost > contourCost and cand2Cost ≥ contourCost then
        /* New seed S = S_{y−1} */
        y = y - 1
    end if
    AddContourLocation(ESS, x, y, contourCost);
end while
```

ESSPointCost(ESS, x, y): This function returns cost of ESS point (x,y). This function queries database to find cost of given point.

AddContourLocation(ESS, x, y, C): This function stores (x, y) as contour location with cost C.

BSearchEdge(d, C): This function performs binary search by varying coordinate value for d'th dimension(other dimensions are fixed) and returns the coordinate value in that dimension at which cost is C.

4.1.3 2-D ESS with multiple contours

The above algorithm that works for a given C (isocost contour) can be extended naturally to identify multiple contours that are in geometric progression on 2D ESS as shown in Figure 4.2(d) where different isocost contours are explored independent of one another.

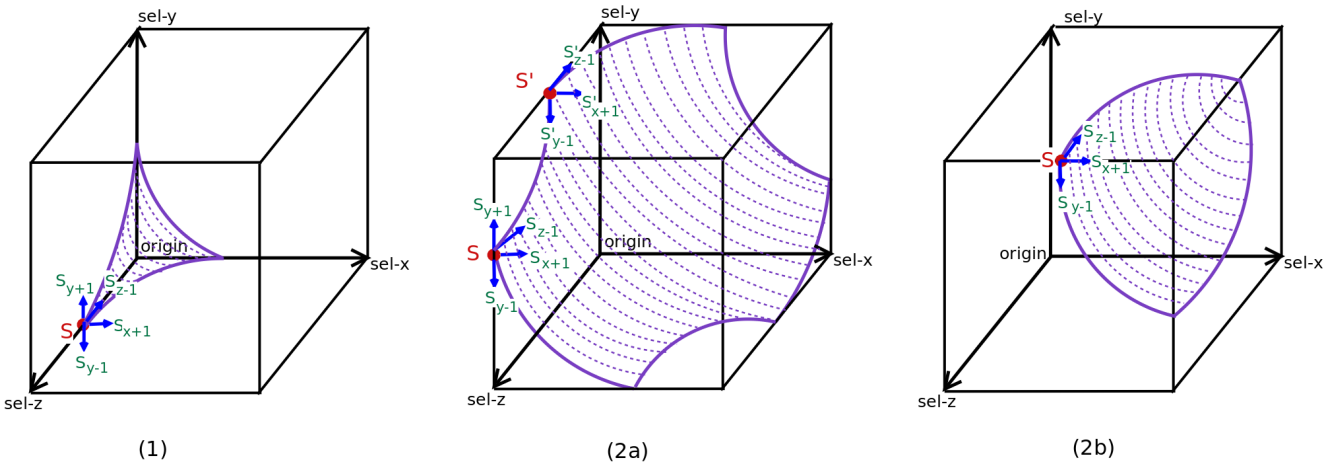


Figure 4.3: Contour Identification in 3D ESS

4.2 Extension to 3D ESS

Next, we show that the neighbourhood exploration approach for contour identification can be extended to general multi-dimensional ESS. For this purpose, we explain the algorithm for 3D ESS that systematically invokes different instances of the 2D algorithm.

4.2.1 Locating the initial Seed

Here, the initial seed (S) is the contour location with maximum z coordinate(In principle, we can start with an initial seed having maximum x coordinate or maximum y coordinate as well). For this purpose, it is first checked whether the seed lies on the boundary $(1, 1, 1)$ to $(1, 1, res)$ which implies $z < res$. If yes, then the seed can be determined by using binary search on the above boundary as shown in shown in case (1) of Figure 4.3. If no, the initial seed is located using a process similar to 2D ESS (for the XY-slice with z = res) which is visualized as case(2) in Figure 4.3 , with two possibilities: (2a) $y < res$ and (2b) $y = res$.

4.2.2 Neighbourhood exploration using Seed

We first explain the contour exploration phase for case (2b). To identify all contour locations with z = res, we can use the 2D exploration algorithm for the XY-slice with z = res and grow the initial seed S as explained in Section 4.1. For exploring the locations with lower values of z, the initial seeds for each XY-slice can be generated by 2D exploration of the XZ-slice (with y = res) with initial seed S and candidate locations S_{x+1} and S_{z-1}. Similarly in case(1), the initial seeds for each lower value of z can be generated by exploring the YZ-slice (with x = 0),

starting with an initial seed S and candidate locations S_{y+1} and S_{z-1} .

Finally, the algorithm for case (2a) proceeds in two sub-phases where the first sub-phase is similar to case(1) until it finds a seed with y = res (shown as S' in case (2a) of the Figure 4.3) and thereafter in second phase it follows algorithm similar to case (2b).

4.3 Impact on bouquet identification overheads

NEXUS algorithm optimizes only that portion of ESS which is close to a given isocost contour while ignoring the remaining space. This reduces the computational overhead to a great extent. It also allows us to quickly identify the first contour and hence start bouquet execution without even exploring locations on later isocost contours. In addition, this approach also makes contour exploration a highly parallelizable task. A new thread can be created whenever a seed is generated for lower dimensional subspace. NEXUS algorithm uses very low memory to perform its task since only the locations on the (accepted) contour are maintained in memory. A thorough empirical analysis of this algorithm is done in Chapter 8.

Chapter 5

Contour-centric Reduction of Plans

Plan Bouquet MSO is dependent on number of plans on the contours. Typically, number of plans on the contours will be very high for any complex query with higher dimensions in ESS. To handle this, earlier, we optimized the entire ESS and reduced plans on the whole ESS using anorexic reduction techniques[3, 4]. Let us call it as *Global Reduction* technique. Though it is computationally intensive, contour plan density is observed to be low after reduction empirically. With the advent of NEXUS algorithm for contour identification, we are restricted to only the plans on the contours as candidates for reduction. In this scenario, we discuss the techniques to reduce contour plan density, considering only the plans on the contours as swallowers of other plans on contours. We measure reduction quality in terms of number of plans on the densest contour.

Each point on the contour is associated with an optimal plan and its cost. Let us consider an ESS location L on the contour with optimal plan 'p'. If a contour plan 'q', when evaluated using FPC at its non-optimal location L, does not increase cost(L) by more than λ(reduction parameter)% then p reduces to q at L. We say 'q' is λ-optimal at L.

All contour locations where 'p' is optimal belong to endo-optimal region of 'p'. Any location L on the contour at which 'p' is not optimal but it does not increase cost(L) by more than λ% than its optimal cost, belong to λ-optimal region of 'p'. Otherwise it belongs to exo-optimal region.

A plan p_j can swallow a plan p_i, if $\forall$ contour points $q \in p_i$(i.e. p_i is optimal at q), $c_j(q)/c_i(q) <= (1 + \lambda)$ where $c_i(q)$ is cost of plan p_i at q.

We first define the *Oracle Reduction* technique as follows.

5.1 Oracle Reduction

In Oracle reduction, all plans on the ESS are candidate swallowers of plans on an individual contour. Given an isocost contour on the ESS, this technique provides the best reduction possible by identifying a minimum size subset of plans in the ESS, that can swallow all plans on the contour. But, it is exponential in complexity and hence not practical. Nevertheless, we will compare the performance of our contour-centric greedy reduction techniques with the oracle reduction by experiments in Chapter 8.

5.2 Contour-centric Reduction

We propose two contour-centric variants of Cost Greedy algorithm with Foreign Plan Costing(FPC) [3, 4] to reduce plans on the contours directly. One performs reduction locally among plans lying on a single contour whereas the other uses plans on all contours to reduce plans on a single contour. FPC refers to estimating the cost of a plan outside its optimality region which is explained in Chapter 6.

5.2.1 Inter-contour Reduction

Reduction is performed using plans on all contours of the ESS together as shown in *Algorithm 2*. Here 'contourPointsSet' contains points lying on a single contour, 'plansSet' is the set of plans on all contours and 'λ' is reduction parameter. If a contour point belongs to more than one plan in $reducedPlansSet$ returned by our algorithm then we pick the plan that results in least cost increase for that point.

5.2.2 Intra-contour Reduction

Here reduction is done on each contour independently to reduce plans lying on that contour. *Algorithm 2* is invoked for each contour independently where 'contourPointsSet' contains points lying on a single contour, 'plansSet' contains set of plans on this contour.

5.2.3 Impact of Contour-centric Reduction

Let res = resolution of ESS, n = total number of plans on ESS, p_i = number of plans on contour 'i' and $|C|$ is the number of contours to find. Then in worst-case, the number of FPC calls made by reduction techniques are compared in Table 5.1.

Inter-contour reduction can not be performed until NEXUS algorithm explores all contours since it considers plans on all contours as candidates for reduction whereas intra-contour reduction can be run in parallel with exploration of subsequent contours by NEXUS algorithm since it requires plans on a single contour only at a time. Also inter-contour reduction uses more

Algorithm 2 Contour Plans Reduction

ReduceContours(contourPointsSet, plansSet, λ)
create n sets $S = \{S_1, S_2, ...S_n\}$ corresponding to n plans in *plansSet* where $S_i = \{L \in contoursPointsSet$ where L is λ-optimal with regard to plan $S_i\}$.

create n sets $E = \{E_1, E_2, ...E_n\}$ corresponding to n plans where $E_i = \{$eating capacity of plan $P_i\}$ calculated using S.
$reducedPlansSet = \phi$
while $contourPointsSet \neq \phi$ **do**
  pick plan p_i from *plansSet* such that $|E_i| = max(|E_j|), \forall E_j \in E$
  $reducedPlansSet = reducedPlansSet \cup p_i$
  $S = S \setminus S_i$
  $E = E \setminus E_i$
  for each $L \in contourPointsSet$ **do**
    if L is λ-optimal with regard to 'p' **then**
      $contourPointsSet = contourPointsSet - L$
    end if
  end for
end while
return *reducedPlansSet*

Algorithm	**FPC Calls**
Global	$n.res^2$
Oracle	$\sum_{i=1}^{\lvert C\rvert} 2.n.res$
Inter-contour	$2.n.res.\lvert C\rvert$
Intra-contour	$\sum_{i=1}^{\lvert C\rvert} 2.p_i.res$

Table 5.1: Number of FPC calls made during reduction in worst-case

memory since we need to maintain information about locations on all contours till the end. Inter-contour reduction results in plans being highly repetitive among neighbouring contours. Therefore, in bouquet execution phase, if we intend to carry forward the work done by a plan in an earlier contour, then inter-contour reduction would be more beneficial. We compare the performance of reduction techniques empirically in Chapter 8.

Chapter 6

Implementation of *ForcePlan* feature

At compile time phase of Plan Bouquet approach, in order to reduce plans on isocost contours, we need to invoke optimizer to cost ESS locations with sub-optimal plans. At run time, we need to do several partial executions of a query with different plans lying on contours. These require database systems to support costing as well as executing a query with different user supplied execution plans. This feature is not supported in PostgreSQL. For this purpose, we have implemented a generic *ForcePlan* feature in latest PostgreSQL database engine.

Definition: Let us call the query plan p_0 that optimizer chooses for a query Q as optimal plan. A query plan p is called a *feasible plan* for Q, if it can be considered as a candidate plan by the typical search strategy of the query optimizer. Any feasible plan for Q that is not optimal is called *foreign plan* for Q. *ForcePlan* feature can force optimizer to choose a user supplied foreign plan for a given query Q. This powerful option provides us with full control over influencing the execution of a query as well since the plan chosen by optimizer is directly passed to executor. The process of costing a foreign plan on an ESS location using *ForcePlan* feature is called *Foreign Plan Costing(FPC)*. Ideally FPC call should take less time compared to optimization since optimizer is not considering all candidate plans before choosing the plan of choice.

Implementation details: *ForcePlan* feature is implemented in PostgreSQL 9.4[10] database engine. It is integrated with *Picasso*[5] Query Optimizer Visualizer tool and tested extensively by forcing plans throughout ESS, including their exo-optimal regions where they are not optimal. It is implemented as a shared library module which can be loaded on demand by PostgreSQL at run time. It is easily portable and the generic interface simplifies using this feature from other applications. We can vary the selectivity of predicates in a query 'Q' and get a corresponding execution plan in XML format using Explain command in PostgreSQL.

Let us call the file that contains the XML plan to be forced as 'plan.xml'. This XML plan can then be forced at a different selectivity location of query Q using *ForcePlan* feature as shown below.

<Query> FPC <plan.xml>

Parser Module: Here input XML file is parsed and information about scans, joins and order of joins are extracted and populated in custom data structures. To parse XML, libxml2[12] C parser library is used.

GUC Configuration: Optimizer behaviour can be controlled through GUC(Grand Unified Configuration) run time configuration parameters of PostgreSQL. They can be used to enable/disable a specific type of scan or join at run time and can be configured at different granularities. We leverage these parameters to implement *ForcePlan* feature by transient assignment of values at a finest granularity i.e. for the duration of a function call.

Base Access Paths: PostgreSQL gets relevant information from catalogs when building access paths for a base relation. At this juncture, we disable all but the scan type to be forced, through GUC transiently for the duration of base access path selection, making the optimizer build only one access path using forced scan type which will essentially be chosen as best path. $IndexOptInfo$ structures need to be cleared to skip index scan based access paths.

Join Access Paths: PostgreSQL builds a $JoinRelOptInfo$ structure for each feasible join combination. It also tries different inner, outer combinations for the given relations. It calls $make\_join\_rel()$ and builds $JoinRelOptInfo$ structure that represents the join of the two given relations. Then PostgreSQL adds it to the path information for paths created with the two relations as inner and outer relation. Our implementation will allow $JoinRelOptInfo$ to be built only for the join combinations occurring in the input XML plan. In this way, the order of joining relations and inner, outer relationships are maintained.

Similarly we have modified $standard\_join\_search()$ method such that only the join types that are to be used to join given 2 relations are allowed at each level through transient setting of GUC parameters. The best path is chosen in normal way and its plan will reflect exactly the forced plan which will be passed to executor. Currently *ForcePlan* feature supports all but nested sub-queries in 'where' clause.

Chapter 7

System Implementation

In this chapter, we mention our contributions in Plan Bouquet prototype system implementation. The overview of existing Plan Bouquet prototype system is shown in Figure 7.1(a). There are two phases in Plan Bouquet approach as mentioned in Chapter 2. In the preprocessing phase, Picasso tool is used to optimize the entire ESS in order to find POSP set, and then reduce the POSP set. The pre-processed data from Picasso is then fed to QUEST, which picks bouquet plans on isocost contours and performs bouquet execution. QUEST is a front-end tool for visualization and bouquet execution. PostgreSQL 8.3 database engine is modified with required features and used in the backend.

We have replaced the bouquet identification approach mentioned above with our proposed bouquet identification techniques. Prior to that, since PostgreSQL 8.3 database engine is obsolete, we have migrated features required for Plan Bouquet technique to latest PostgreSQL

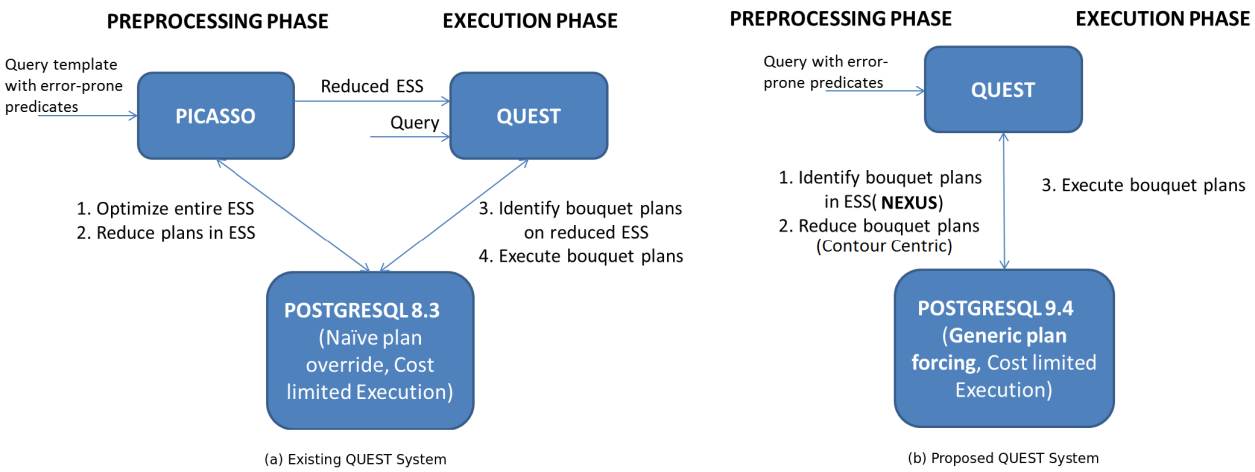


Figure 7.1: QUEST System Overview

database engine. The following are the major enhancements and feature additions done in PostgreSQL 9.4 database engine and QUEST front-end tool.

7.1 Modifications in database engine

- We have implemented *ForcePlan* feature with a generic interface in PostgreSQL 9.4 database engine as explained in Chapter 6.
- We have migrated the following features implemented in PostgreSQL 8.3 database engine for Plan Bouquet execution like
  - cost monitoring
  - cost bounded plan execution
  - sub-plan execution
  - selectivity monitoring

  to *PostgreSQL 9.4* database engine.

7.2 Modifications in prototype system

- We have implemented and integrated the *NEXUS contour identification* algorithm proposed in Chapter 4 and *contour-centric plan reduction* techniques proposed in Chapter 5 with QUEST system.
- We have enhanced QUEST front-end tool to use our *ForcePlan* feature proposed in Chapter 6 during bouquet identification and bouquet execution.
- We have used GNUPlot[13] for visualization of 2D and 3D diagrams since it is lightweight and powerful. We have also improved the GUI aspects of QUEST front-end.

The overview of QUEST system with our enhancements is shown in Figure 7.1(b). We have evaluated Plan Bouquet prototype system over TPC-H and TPC-DS benchmark environments.

Chapter 8

Experimental Evaluation

We now turn our attention towards analysing the performance of our contour identification algorithm and contour-centric plan density reduction techniques on a variety of complex queries. Time overhead and memory space overhead are the major evaluation criteria for contour identification. Memory is critical since maintaining contour locations in memory during preprocessing is a huge overhead in higher dimensions. We compare and contrast the performance of different techniques to reduce contour plans in detail with various experiments. Finally we show how the bouquet identification phase works with our contour identification and contour-centric reduction techniques incorporated in Plan Bouquet prototype system.

The database engine used in our experiments is a modified version of PostgreSQL 9.4[10], incorporating the changes outlined in Section 7.1. The hardware platform is a vanilla Sun Ultra 24 workstation with 8 GB of memory and 728 GB of hard disk. Experiments are conducted with TPC-H 1GB database using TPC-H benchmark queries, and TPC-DS 100GB database using TPC-DS benchmark queries[11].

8.1 Efficient Contour Identification

Experimental results for identifying isocost contours using our NEXUS algorithm in 2D and 3D ESS are presented here followed by an analysis of overheads. Inter contour cost ratio $r = 2$ is used throughout this Section.

8.1.1 2D ESS

Example 1: TPC-H Query Template 5, ESS Dimensions: 2, ESS Resolution: 300

There are totally 8 contours on the 2D ESS for Example 1. Contours produced by NEXUS algorithm are shown in Figure 8.1(a) where each isocost contour is denoted with a unique colour. Figure 8.1(b) shows optimizer chosen plans on isocost contours where each colour denotes a

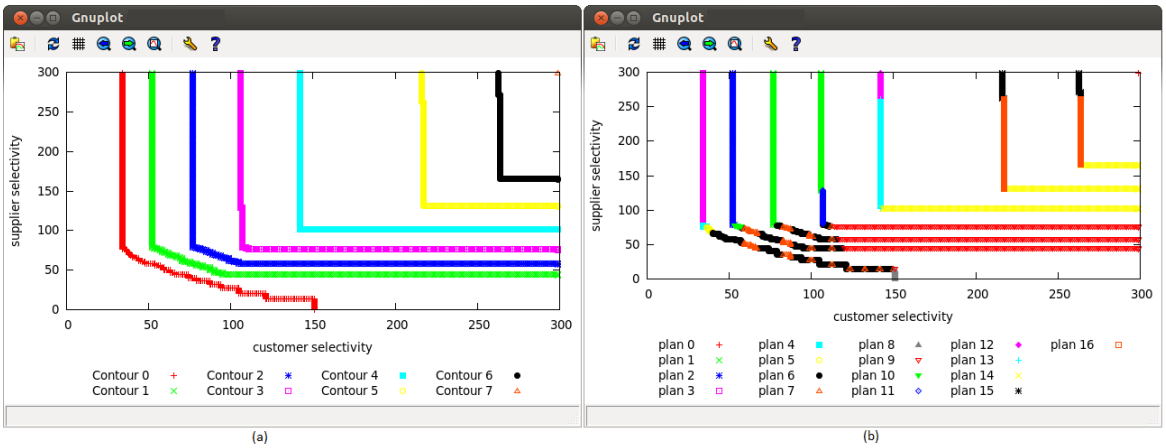


Figure 8.1: Isocost Contours for Example 1

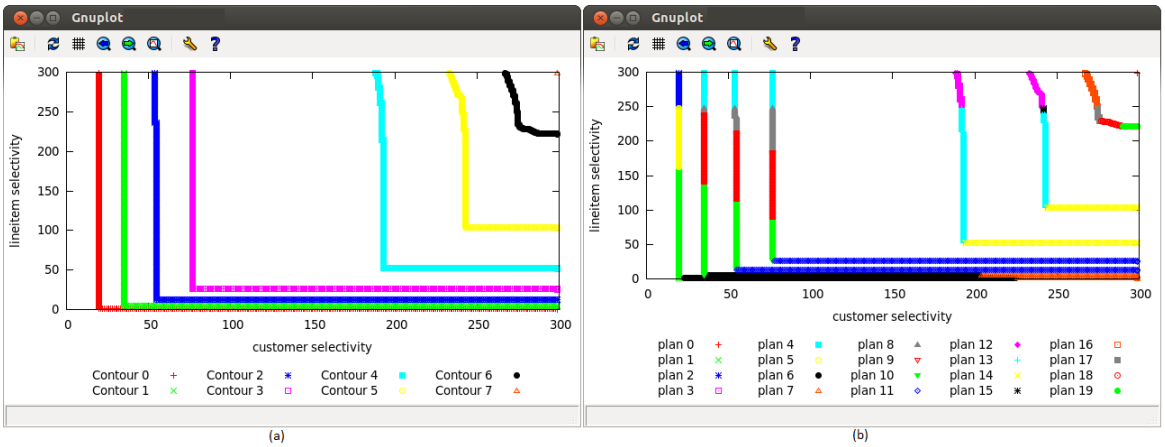


Figure 8.2: Isocost Contours for Example 2

unique plan in the ESS.

Example 2: TPC-H Query Template 10, ESS Dimensions : 2, ESS Resolution : 300

There are totally 8 contours on the 2D ESS for Example 2. Contours produced by NEXUS algorithm are shown in Figure 8.2(a) where each isocost contour is denoted with a unique colour. Figure 8.2(b) shows optimizer chosen plans on isocost contours where each colour denotes a unique plan in the ESS.

8.1.2 3D ESS

Example 3: TPC-H Query Template 8, ESS Dimensions : 3, ESS Resolution : 100

There are totally 7 contours on the 3D ESS for Example 3. Contours produced by NEXUS algorithm are shown in Figure 8.3 where each isocost contour is denoted with a unique colour.

Table 8.1 compares the number of optimization calls made by the algorithms where 'Global'

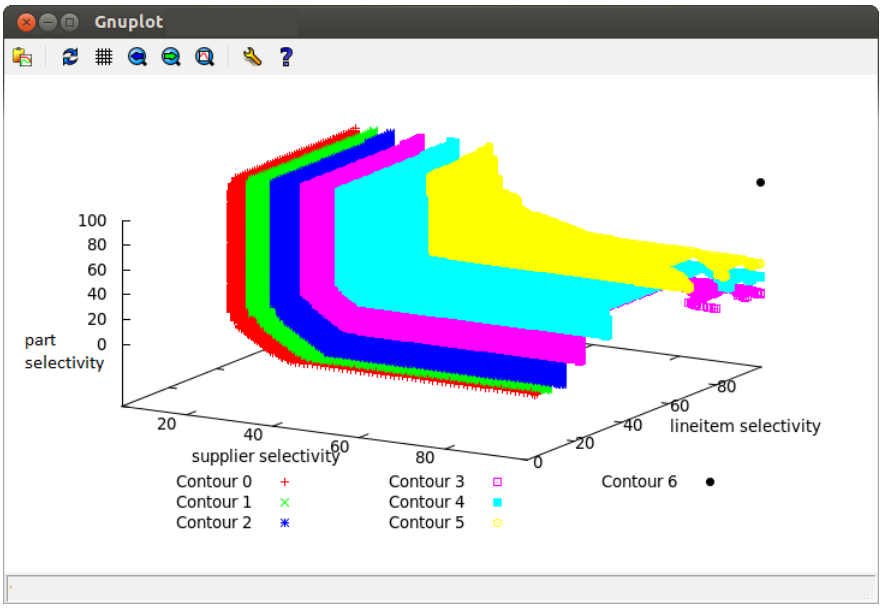


Figure 8.3: Isocost Contours for Example 3

denotes the existing bouquet identification technique which optimizes the entire ESS, 'Ideal' denotes the ideal case and 'NEXUS' denotes the NEXUS algorithm.

Example #	Global	Ideal	NEXUS
1	90000	2585	5220
2	90000	2904	5856
3	1000000	76300	152660

Table 8.1: Number of optimization calls made

Note that the NEXUS algorithm performs twice the number of optimization calls as compared to Ideal case, barring the small number of optimization calls made to locate the initial seed for each contour.

8.1.3 Time and Space overhead

To understand the magnitude of savings we get in terms of time and space, refer to Figure 8.4 and Figure 8.5. The time overheads of existing technique('Global') and NEXUS algorithm are shown in minutes on a log-scale in Figure 8.4. Similarly, their space overheads are shown in KBs on a log-scale in Figure 8.5.

The terminology used for the queries is wD_x_Qy_z, where w indicates the number of dimensions, x the benchmark (H or DS), y the query number in the benchmark and z the ESS resolution(optional). So, for example, 2D_H_Q5_300 indicates a two-dimensional ESS of resolu-

tion 300 on Query 5 of the TPC-H benchmark. For a query with 5 dimensions in ESS(resolution = 50), the projected requirements are 625 hours of time and 4.7GB of memory for existing technique against 28 hours and 143MB required for NEXUS algorithm.

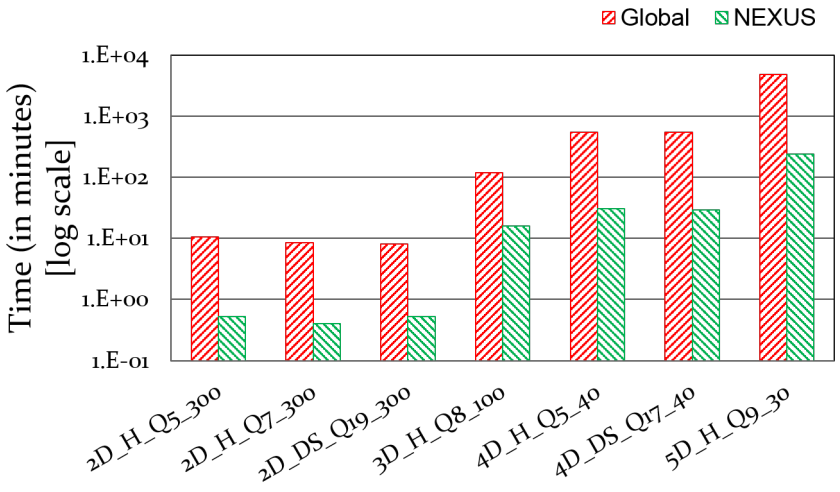


Figure 8.4: Contour Identification: Time Overhead

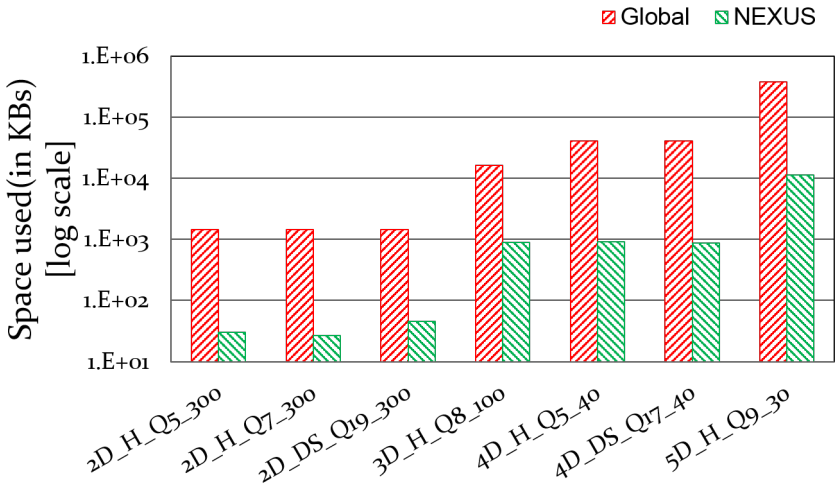
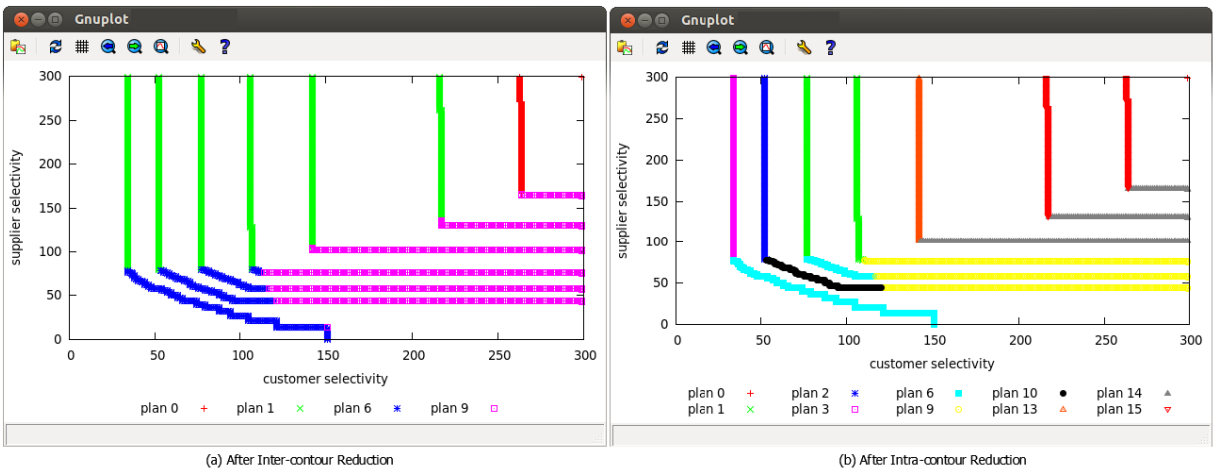


Figure 8.5: Contour Identification: Space Overhead

8.2 Contour-centric Reduction of Plans

In this section, we analyse how reducing plans on the contours identified by NEXUS algorithm performs in comparison with reducing plans on the contours identified by optimizing the entire ESS and finding the complete POSP set as well as oracle plan reduction that we introduced in



(a) After Inter-contour Reduction (b) After Intra-contour Reduction

Figure 8.6: Contours plans after reduction for Example 1

Chapter 5. The reduction quality is measured in terms of number of plans on the densest contour since plan bouquet MSO is directly dependent on this. We have used reduction parameter λ = 20% and ESS resolution of 300 for 2D, 30 for 3D, 20 for 4D and 10 for 5D throughout this section.

8.2.1 Reduced Contour Plan Density

Figure 8.6(a) shows contour plans after inter-contour reduction and Figure 8.6(b) shows contour plans after intra-contour reduction for Example 1. Each colour denotes a unique plan in ESS. As can be seen, after intra-contour reduction there is less repetition of plans among contours but after inter-contour reduction there is a high repetition of plans among neighbouring contours.

Table 8.2 shows the number of plans per contour for Example 3(with an ESS resolution of 30), before reduction and after each of the reduction technique is applied. Here, 'Global' is the existing technique used for contour plan density reduction, wherein the entire ESS is optimized first. Then anorexic reduction is performed on the entire ESS to identify reduced contour plan set. 'Inter-contour' is the inter-contour reduction technique and 'Intra-contour' is the intra-contour reduction technique and 'Oracle' is the oracle reduction technique. As shown in the figure, contour-centric reduction techniques perform on par with global reduction technique and oracle reduction as well.

8.2.2 Plan Bouquet MSO Bound

Figure 8.7 compares reduction techniques in terms of plan bouquet MSO bound, which is dependent on number of plans on the densest contour.

The reduction quality of contour-centric plan reduction techniques is on par with oracle

Contour #	Plans Count				
	Before Reduction	Global	Oracle	Inter-contour	Intra-contour
1	36	3	3	3	3
2	34	4	4	4	4
3	33	4	4	4	4
4	58	5	4	4	4
5	33	5	3	4	3
6	30	4	4	4	4
7	1	1	1	1	1

Table 8.2: Contour-wise plans count for Example 3

plan reduction and better than global reduction technique.

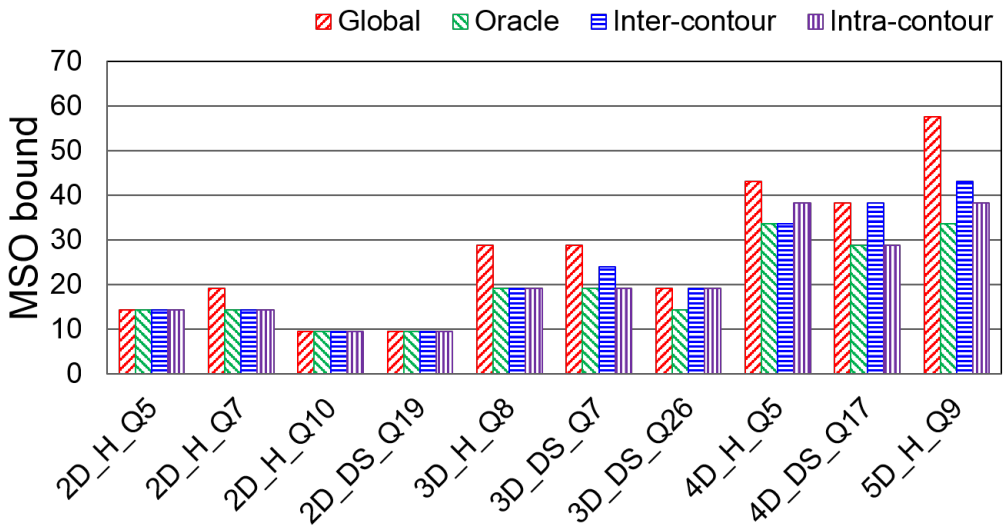


Figure 8.7: MSO bound

8.2.3 FPC Calls

Figure 8.8 compares number of FPC calls made by these algorithms. Though query optimization is not performed for an FPC call, it is still an overhead since it invokes the optimizer for each call. Hence, lesser the number of FPC calls, lesser the overhead. Among the new schemes, number of FPC calls made by intra-contour reduction is always low when compared with inter-contour reduction since reduction is local to single contour.

It is evident from the above experiments that the contour-centric reduction techniques perform well in comparison with full-blown reduction technique. Their performance is also

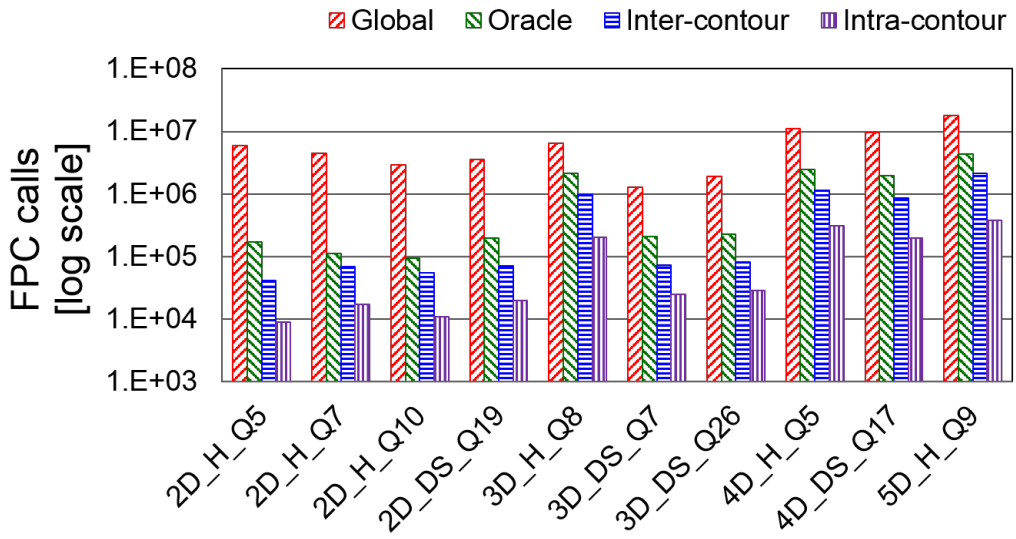


Figure 8.8: FPC calls made during reduction

comparable with oracle reduction as far as Plan Bouquet reduction quality is concerned. Among the contour-centric reduction techniques, intra-contour reduction makes lesser number of FPC calls, uses less memory space and still provides reasonably good reduction quality.

It is highly beneficial to use our contour identification algorithm in order to identify isocost contours directly and then utilize our contour-centric reduction techniques to get reduced bouquet plans. In this way, we achieve good MSO in plan bouquet approach with enormous savings in time and memory space, which is very critical when ESS dimensionality is higher.

8.3 Plan Bouquet prototype system

Our proposed bouquet identification techniques are integrated within Plan Bouquet prototype system(QUEST). As an example, TPC-H Query 5 with 2D ESS is given as input to QUEST and its graphical display for bouquet identification is shown in Figure 8.9. Here, *Contour plan diagram*, which is on the left side of the figure shows isocost contours identified by our contour identification algorithm with plans. There are totally 8 contours in the ESS. The contour plans are then subjected to inter-contour plan reduction. The reduced plans on the contours are shown in *Reduced contour plan diagram*, on the right side of Figure 8.9. Contour-wise plan details and MSO guarantees are shown in lower right side of the figure.

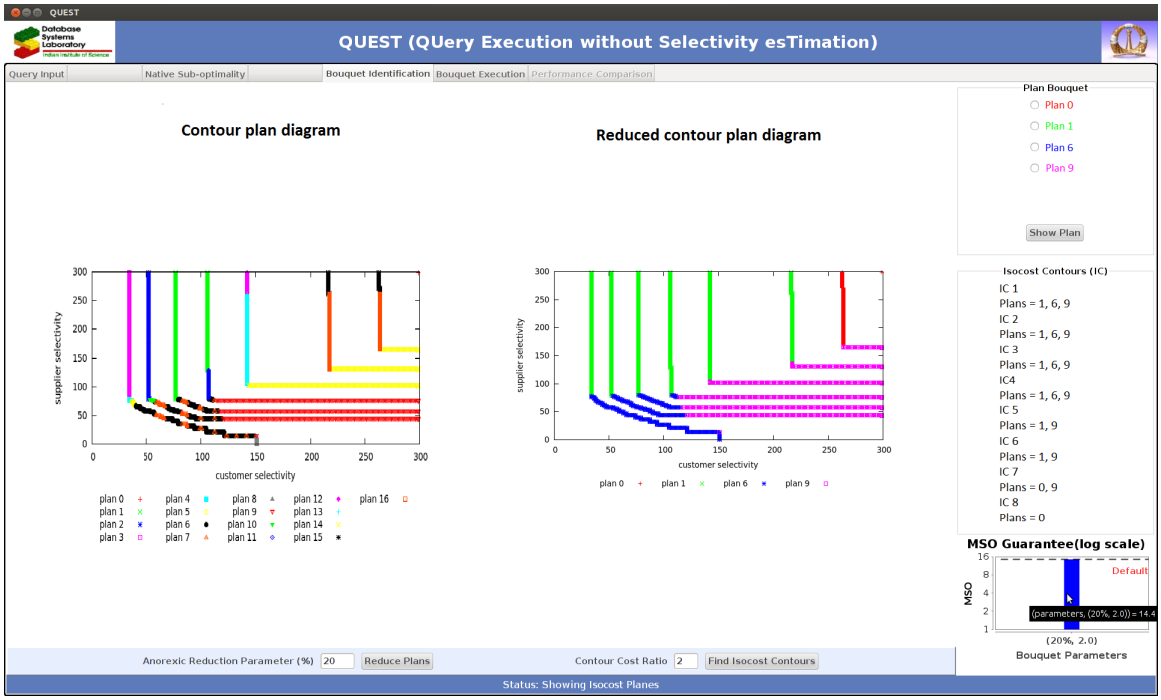


Figure 8.9: Bouquet Identification Interface in QUEST

Chapter 9

Conclusion and Future Work

In this work, we have proposed efficient techniques to identify bouquet plans for Plan Bouquet approach. Specifically, we introduced an efficient algorithm to identify isocost contours on a multi-dimensional ESS and explored contour-centric variants of plan reduction technique. Also, a generic *ForcePlan* feature has been implemented in PostgreSQL database engine. Our bouquet identification techniques are integrated within Plan Bouquet prototype system and evaluated on PostgreSQL 9.4 over TPC-H and TPC-DS benchmark environments. The experimental evaluation exhibits that our bouquet identification techniques reduce the preprocessing overhead of Plan Bouquet approach to a great extent. Further, the reduction quality is shown to be comparable to oracle reduction technique, thereby preserving the performance guarantees.

It would be an interesting future work to explore whether it is feasible to identify plans on the contour without explicitly optimizing all the contour locations since many of them return identical plans.

Bibliography

[1] A. Dutt and J. Haritsa, "Plan Bouquets: Query Processing without Selectivity Estimation", *SIGMOD*, 2014. http://dsl.serc.iisc.ernet.in/publications/conference/bouquet.pdf.

[2] A. Dutt, S. Neelam and J. Haritsa, "QUEST: An Exploratory Approach to Robust Query Processing", *PVLDB*, 2014. http://dsl.serc.iisc.ernet.in/projects/QUEST/.

[3] Harish D., P. Darera and J. Haritsa, "On the Production of Anorexic Plan Diagrams", *VLDB*, 2007.

[4] Harish D., P. Darera and J. Haritsa, "Identifying Robust Plans through Plan Diagram Reduction", *VLDB*, 2008.

[5] J. Haritsa, "The Picasso Database Query Optimizer Visualizer", *VLDB*, 2010.

[6] M. Stillger, G. Lohman, V. Markl and M. Kandil, "LEO - DB2's LEarning Optimizer", *VLDB*, 2001.

[7] A. Deshpande, Z. Ives and V. Raman, "Foundations and Trends in Databases", 2007.

[8] David Gries, "The Science of Programming", 1st ed., Springer-Verlag New York, Inc., 1987.

[9] http://www.postgresql.org/docs/8.3/static/index.html.

[10] http://www.postgresql.org/docs/9.4/static/index.html.

[11] http://www.tpc.org.

[12] http://xmlsoft.org/index.html.

[13] http://www.gnuplot.info.