Improving Worst-case Bounds for Plan Bouquet based Techniques

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

Faculty of Engineering

BY

Lohit Krishnan

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

June, 2015

Declaration of Originality

I, **Lohit Krishnan**, with SR No. **04-04-00-10-41-13-1-10178** hereby declare that the material presented in the thesis titled

Improving Worst-case Bounds for Plan Bouquet based Techniques

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2013-2015**.
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 discusions.
- 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

Acknowledgements

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

I am thankful to Srinivas Karthik and Anshuman Dutt for the numerous engaging discussions we had in lab. Their suggestions helped me a lot. It was a great experience, working with them. This project would not have been possible without their constant support and motivation.

My sincere thanks goes to my fellow lab mates for all the help and suggestions. Also I thank my CSA friends, especially *'Asylum'* who made my stay at IISc very memorable. Finally, I am indebted with gratitude to my family for their love and inspiration that no amount of thanks can suffice.

Abstract

Given an SQL query, current database systems execute it using a least cost plan which is largely based on estimates of predicate selectivities. Due to insufficient statistics and invalid assumptions, errors in estimates can lead to highly sub-optimal plans.

In the paper[1], a strategy named "Plan Bouquets" has been proposed which provides guarantees on the worst case execution performance which does not rely on the estimates of predicate selectivities. The `PlanBouquet` algorithm, in its basic form, *implicitly* discovers predicate selectivities by observing the completion status of a sequence of cost-budgeted plan executions.

Our contribution includes improving two variants of `PlanBouquet`. In the first contribution, we analyze this "non-intrusive" bouquet technique in presence of assumptions on the acclivities of cost functions which generally hold in practice. Further, we show that we can achieve significant reduction in the preprocessing time and will get upper bound on worst-case performance which is independent of the plan densities.

Next, we investigate an intrusive variant of `PlanBouquet` named `SpillBound`[2], which changes the plan execution component and gives worst-case performance bound of $O(D^2)$, which is only dependent on D, the dimensionality of selectivity space. We propose `Opt-SB`, which dynamically optimizes `SpillBound`, such that, the worst-case bound oscillates between $O(D)$ and $O(D^2)$ based on the optimizer's behaviour profile within the selectivity space.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

2 Problem Framework **3**

2.1 Error-prone Selectivity Space (ESS) 3
2.2 Maximum Sub-optimality (MSO) 4
2.3 Problem Definition 5
2.4 Plan Bouquet Approach 5
2.4.1 1D Plan Bouquet 5
2.4.2 Extension to Multidimension 7

3 Improving Non-intrusive technique **9**

3.1 Proposed Solution 10
3.1.1 Finding the covering set 11
3.2 Theoretical Results 13
3.2.1 Preprocessing time incurred 13
3.2.2 Impact on MSO Bound 14
3.3 Experimental Evaluation 15

4 Improving intrusive technique **18**
4.1 SpillBound Algorithm [2] 18
4.1.1 The `2D-SpillBound` Algorithm 20
4.1.2 Extending to higher dimensions 21
4.2 Proposed Solution 22
4.2.1 Favourable Case 23
4.2.2 `Opt-SB` 26
4.3 Results 30
4.3.1 Experimental Setup 30
4.3.2 MSO Bound 31
4.3.3 Empirical MSO 31
4.3.4 Average-case Performance 31

5 Conclusion **33**

Bibliography **34**

List of Figures

2.1 Plan Bouquet with single dimension `ESS` 6
2.2 Contours in 2D `ESS` 7

3.1 Covering Set 11
3.2 Comparison of the number of optimizations 13

4.1 Spilled Plans 19
4.2 Pruning of ESS (`PlanBouquet` and `SpillBound`) 20
4.3 Favourable case for `SpillBound` 22
4.4 Big Picture 25

List of Tables

2.1 Notations 8

3.1 Percent violation of *cost acclivity assumption* 17
3.2 MSO Bounds of $\mathcal{P}$' for 2D ESS 17
3.3 Empirical MSO 17

4.1 Minimum tolerance needed to align all contours. 25
4.2 Comparison of query specific MSO Bound 31
4.3 Empirical MSO using *local-partitioning* 32
4.4 Empirical values of t and p 32
4.5 Empirical ASO using *local-partitioning* 32

Chapter 1

Introduction

Modern cost-based database query optimizers estimate a host of predicate selectivities while identifying the least cost plan for a declarative query. Often, the selectivity estimates are significantly erroneous with respect to the actual values subsequently encountered during query execution. Such errors lead to poor execution plan choices by the optimizer, resulting in substantially inflated query response times.

In the efforts to mitigate this problem, [1] proposed a new non-intrusive query processing strategy called "Plan Bouquets", which provides upper bound on worst-case execution performance. The basic idea in the bouquet approach is to completely jettison the compile-time estimation process for error prone selectivities. Instead, these selectivities are discovered at run-time through a sequence of cost-limited executions from a small set of plans. A potent benefit of this discovery-based approach is that it lends itself, for the first time in the literature, for providing *guaranteed bounds on worst-case optimizer performance*. Specifically, if we compute **MSO** (Maximum Sub-Optimality) as the worst-case ratio, over the entire selectivity space, of the cost sub-optimality incurred by the optimizer with respect to an oracular system that magically knows the correct selectivity values, then the plan bouquet can provide guaranteed upper bounds on MSO – specifically, $MSO \leq 4 * |PlanBouquet|$. [1] Moreover, the worst case execution guarantee is dependent on the plan density behaviour.

We investigate a scenario, when the plan cost functions will adhere to certain desirable functional properties. We observed that, in practical scenarios, plan cost functions obey a specific property which we term as "cost acclivity assumption" which will be described in Chapter 3. As our initial contribution, we will utilize this assumption to improve the existing MSO bound given by `PlanBouquet`. Moreover, we will also look at the resulting benefits on

[1] A more precise bound is given in Chapter 2.

the preprocessing time.

Recently, [2] proposed an improvement to the above technique named "SpillBound", which improves the MSO bounds by using the power of intrusiveness into the database engine. Specifically, they leverage the notion of *"spilling"*, whereby operator pipelines in the execution plans are prematurely terminated at carefully chosen locations in the plan tree. The use of spilling is tuned towards ensuring that the assigned budgets for plan executions are selectively focussed on speeding up the learning process.

Our second contribution includes improving the MSO bounds provided by SpillBound. We propose an optimized version of SpillBound, named Opt-SB, where we execute plans at strategic selectivity locations to maximize the selectivity learning given by SpillBound. The distinctive feature of Opt-SB includes execution of plans that might be slightly sub-optimal which gives better selectivity learning than SpillBound. The execution of plans is done after partitioning D dimensions of selectivity space into p partitions. We finally give a MSO bound which is in $O(Dp)$. As an example, for TPC-DS query 19 with 5 error-prone predicates, MSO bound given by SpillBound is 40 while Opt-SB brings it down to 12.6.

Organization

The rest of the thesis is organized as follows: In Chapter 2, a precise description of the robustness model is provided, along with the associated notations. We also define the problem statement and give a brief background of PlanBouquet in Chapter 2. In Chapter 3, we try to give an MSO bound for the non-intrusive system by using cost acclivity assumption. In Chapter 4, we briefly describe the intrusive variant of PlanBouquet, namely SpillBound and propose an optimized version of it named Opt-SB. Formulation of MSO Bound for the optimized version and it's associated empirical results are explained in Chapter 4. We conclude our work in Chapter 5.

Chapter 2

Problem Framework

In this chapter, we present the robustness model used in this thesis, and the key notations and concepts, followed by a brief overview of the plan bouquet approach.

While different notions of robustness are relevant for different scenarios, we use here the following measure, introduced in [1]: Robustness is evaluated in terms of the sub-optimality of the overall execution cost in comparison to the optimal cost incurred by an oracle that possesses complete a priori knowledge of all predicate selectivities.

2.1 Error-prone Selectivity Space (ESS)

Consider a query for which a subset of the predicate selectivities cannot be estimated accurately. We call one such error-prone predicate as `epp`, and a collection of these as `EPPs` (equivalently, `epp` set). All possible selectivity combination of the `EPPs`, constitute the error-prone selectivity space, i.e, `ESS`, whose dimensionality is denoted by D. The `ESS` is represented by a discretized grid with the values on each dimension ranging over $[0, 1]$. The `epp` corresponding to dimension j of the `ESS` is denoted by R_j.

Each location $q \in [0, 1]^D$, represents a unique query in the `ESS`, with $q.j$ denoting the selectivity of R_j. For example, consider an `ESS` of dimension 2 and a location $q = (0.3, 0.2) \in [0, 1]^2$. In our notation, $q.1$ would be 0.3 and $q.2$ would be 0.2, representing selectivity instance of the corresponding two error-prone predicates. Let $\succ$ denote a binary relation on the set of selectivity locations over the entire ESS, which is defined as follows.
For any two locations q_b and q_c in `ESS`, we say $q_b \succ q_c$
if $q_b.i > q_c.i \quad \forall i \in \{1, \ldots, D\}$

For a given query and a location in the `ESS` (thus fixing selectivity for all the `EPPs`), the query optimizer can identify the optimal query execution plan. Therefore, at the time of compiling the query, one can identify the optimal plan for each location in the ESS grid. This can be done by repeated invocations of the optimizer, and explicit injection of selectivities. The optimal execution plan for a location q is denoted by P_q, and $Cost(P_q, q')$ represents the cost of executing the query incurred by the plan P_q when the actual selectivities coincide with $q' \in$ ESS. Therefore, $Cost(P_q, q)$ represents the optimal execution cost of the query when the run-time selectivities correspond to q (For the clarification of the reader, we alternatively denote $Cost(P_q, q)$ as $OC(q)$). The set of plans that cover the ESS space constitutes the Parametric Optimal Set of Plans (POSP) [3]. The locations in the ESS for which a POSP plan P is optimal are collectively referred to as the "endo-optimal region" of plan P [4].

We adopt the convention of using q_a to denote *actual* run-time selectivities. For optimizers that execute a single selected plan, it first estimates a selectivity location for the query and then the plan to be executed is chosen based on it. Let, q_e denote the single *estimated* selectivity location decided by the optimizer. However, for plan switching-based schemes like PlanBouquet, a *sequence* of locations are explored. This sequence is called a "run". Further, the running selectivity location, as progressively discovered by the bouquet mechanism, is denoted by q_{run}.

2.2 Maximum Sub-optimality (MSO)

We now present the key notion of sub-optimality used as the measure of robustness in [1]. Consider the POSP plan P_{q_e}, representing the optimal plan at location $q_e \in$ ESS. The sub-optimality of using P_{q_e} when the actual selectivity turns out to be q_a is given by

$$SubOpt(q_e, q_a) = \frac{Cost(P_{q_e}, q_a)}{Cost(P_{q_a}, q_a)} \quad \forall (q_e, q_a) \in \texttt{ESS} \tag{2.1}$$

The quantity $SubOpt(q_e, q_a)$ ranges over $[1, \infty)$. Now, the Maximum Sub-Optimality (MSO) over the entire ESS is given by

$$MSO = \max_{(q_e, q_a) \in \texttt{ESS}} (SubOpt(q_e, q_a)) \tag{2.2}$$

The above definition is suitable for a traditional query processing engine where a single plan is used to execute a query. However, in case of plan switching approaches such as PlanBouquet, multiple plans are executed in a cost-limited manner. We represent each such algorithm by a sequence of $(plan, budget)$ pairs. Since the sequence changes for each $q_a \in$ ESS, we call such a sequence, with respect to a q_a, by $\texttt{Run}_{q_a}$. Thus, in this case the suboptimality incurred for a given q_a, denoted by $SubOpt(*, q_a)$, is given by

$$SubOpt(*, q_a) = \frac{\sum\limits_{(plan,budget) \in \texttt{Run}_{q_a}} budget}{Cost(P_{q_a}, q_a)}.$$

Further, we define MSO as

$$MSO = \max_{q_a \in \texttt{ESS}} SubOpt(*, q_a) \tag{2.3}$$

Average Sub-Optimality(ASO) Similar to worst case impact, we also define a metric for evaluating average case. Formally, the average-case equivalent definition of MSO is the following:

$$ASO = \frac{\sum\limits_{q_a \in \texttt{ESS}} SubOpt(*, q_a)}{\sum\limits_{q_a \in \texttt{ESS}} 1} \tag{2.4}$$

2.3 Problem Definition

Within the above framework, the problem of robust query execution is defined as:

Given a user query Q with D error-prone predicates, and the ESS *populated with the POSP plans, develop a query processing approach that will give an upper bound on MSO.*

The key assumptions that allow us to systematically explore the ESS are those of *plan cost monotonicity* and *selectivity independence*. They may be stated as:

Assumption 2.1. *Plan Cost Monotonicity (PCM): For any two locations $q_b, q_c \in$ ESS, and for any plan P,*

$$q_b \prec q_c \Rightarrow Cost(P, q_b) < Cost(P, q_c) \tag{2.5}$$

Assumption 2.2. *Selectivity Independence: The selectivities of the error-prone predicates are independent with respect to each other.*

2.4 Plan Bouquet Approach

The plan bouquet approach [1] systematically discovers the actual selectivities at run-time through a sequence of cost-limited executions of small set of plans. To understand this concept, let us start with the case of single epp (referred to as 1D Plan Bouquet), and then move on to multiple epp scenario.

2.4.1 1D Plan Bouquet

A single epp induces an 1D ESS by varying selectivities of the epp as shown in Figure 2.1. Here, the x-axis captures the selectivity range of the epp, while the y-axis denotes the plan

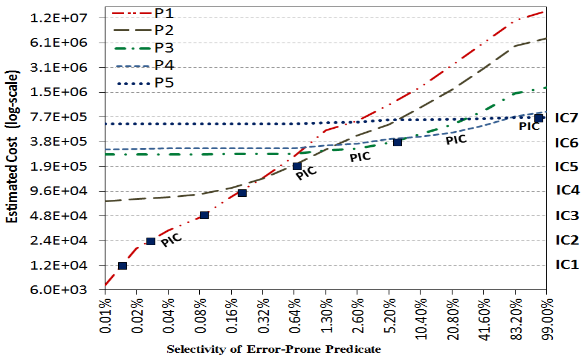

Figure 2.1: Plan Bouquet with single dimension ESS

execution costs corresponding to each selectivity value. There are five POSP plans, and cost variation for each of them can be seen from the figure. Further, it is evident that each one of them is the least cost (or best) plan over disjoint selectivity segments. This pointwise minimum cost curve among all the plans at each of the locations in the ESS is referred to as the *POSP Infimum Curve* (PIC). In this case, the PIC is a one-dimensional curve, whereas for the general case with D epp, the PIC is a D-dimensional surface. For each location $q \in$ ESS, PIC satisfies the invariant property that it indicates the optimal cost of executing the query if the run-time selectivities coincide with q.

Now, we introduce the notion of isocost contour (IC) which the plan bouquet approach is predicated on. *The isocost contour of cost C is the set of locations in the* ESS *whose PIC cost is equal to C.* Since the PIC cost is monotonically increasing with selectivity, the isocost contours are singleton points (in case of single epp). For instance in the example, we can see that the isocost contours $IC1, \ldots, IC7$ intersect the PIC at singleton points which are marked. For a PIC whose minimum cost is C_{min} and maximum cost is C_{max}, plan bouquet approach is selectively interested in those isocost contours whose cost is of the form $2^k \cdot C_{min}$ for all $k = 1, \ldots, \lceil \log_2(\frac{C_{max}}{C_{min}}) \rceil$ – that is, a *contour cost-doubling regime* is in operation. This small set of plans on all the isocost contours are called as “plan bouquet”.

1D Plan Bouquet Execution: Now we shall see the execution strategy of the bouquet of plans, which are identified at compile time. Starting from the cheapest isocost contour, one plan is executed in each contour with budget equal to cost of the contour, until a plan completely finishes its execution. In our example, let us say that the actual selectivity of the epp is 5% i.e. $q_a = 5\%$. To begin with, plan P1 is executed with budget equal to cost corresponding to the cheapest isocost step IC1. Since the budget does not suffice (which is inferred when the plan does not finish executing completely), we increase the budget until we reach IC4, after

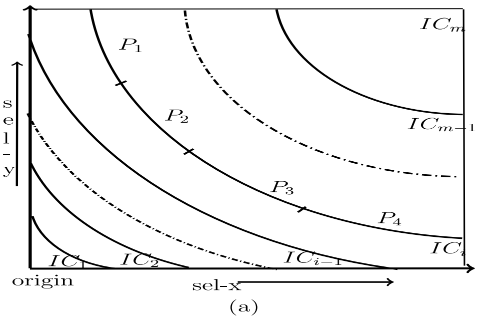

Figure 2.2: Contours in 2D ESS

unsuccessful executions with budgets corresponding to IC2 and IC3, continuing execution of the same plan. Since none of the previous four executions complete, the plan is changed to P2 with increased budget of IC5. This execution again does not go to completion. Finally, execution of P3 with budget of IC6 finishes completely, since the actual location, 5%, is within the selectivity range covered by IC6. Further, it can shown that using this approach, for 1D case, the MSO to be at most 4.

2.4.2 Extension to Multidimension

As mentioned before, the PIC, for ESS with D dimensions, is a D-dimensional surface. Similarly, the isocost contour translates to a continuous surface of dimension $D-1$. For example, Figure 2.2 shows the hyperbolic isocost contours that result with a 2-dimensional ESS. Further in each isocost contour, we see that there is more than one plan, each associated with disjoint regions of the contour. In the example, plans P_1, P_2, P_3, P_4 are the optimal plans for different regions of contour IC_i. This set of plans associated with a contour are collectively referred to as $\texttt{PL}(IC_i)$, and the cost of the contour is denoted by $\texttt{CC}(IC_i)$.

Plan Bouquet Execution: Let us assume that there are m cost-doubling contours $IC_1, \ldots, IC_m$ where $m = \lceil(\log_2 \frac{C_{max}}{C_{min}})\rceil$. Now, Starting from IC_1, *all* plans in the contour are executed with the contour cost budget, until one finishes its execution. For instance, in contour IC_i, either one of the plan, $P_1, \ldots, P_4$, completes execution within the assigned budget – in which case the query is answered – or, if none of the plans complete, the search proceeds to the next contour IC_{i+1}. Thus, the MSO for this algorithm is captured in the following theorem.

Theorem 2.1. *[1] The* `PlanBouquet` *algorithm has an MSO bound of* 4ρ *where* ρ *is the maximum number of plans in any contour, i.e.,* $\rho = \underset{i=\{1,\ldots,m\}}{Max}\{|\texttt{PL}(IC_i)|\}$.

So we can infer that the MSO bound given by `PlanBouquet` is dependent on the plan density behaviour of the contours in the ESS. Theoretically, ρ value could as large as the cardinality of the POSP set, which may be huge. In the next chapter, we will investigate this problem and address it by utilizing a property on PIC.

For easy reference, the notations discussed so far, and those used in the following chapters, are summarized in Table 2.1.

Notation	**Meaning**
epp (EPPs)	Error-prone predicate (its collection)
ESS	Error-prone selectivity space
D	Number of dimensions of ESS
$R_1, \ldots, R_D$	D error prone predicates
$q \in [0,1]^D$	A location in the ESS space
$q.j$	Selectivity of q in the jth dimension of ESS
P_q	Optimal Plan at $q \in$ ESS
q_a	Actual run-time selectivity
q_{run}	The running selectivity location, as progressively discovered by SpillBound and Opt-SB
$Cost(P, q)$	Cost of plan P at location q
$OC(q)$	Cost of optimal plan at location q
IC	Isocost Contour
res	Resolution of ESS grid
CC(IC)	Cost of an isocost contour IC
PL(IC)	Set of plans on contour IC
Int(P)	Set of non-leaf nodes of plan P
P^j	Plan P is identified to spill on epp R_j

Table 2.1: Notations

Chapter 3

Improving Non-intrusive technique

In this work, we try to investigate the properties of plan functions and utilize their characteristics to improve `PlanBouquet`. Although `PlanBouquet` give worst case execution guarantees, the expression for MSO depends on ρ, which is indirectly dependent on the nature of the plan diagram. Theoretically, ρ could be as large as the number of plans in the plan diagram, hence we investigate whether we can give a MSO bound independent of plan diagram characteristics. Moreover, a huge amount of preprocessing is required for the basic plan bouquet to get the bouquet of plans. Again, by utilizing a distinctive property of PIC, we try to reduce the preprocessing overheads. We will start by stating the assumption which we term as "cost acclivity" and later see its effect on both guarantees and the preprocessing time.

Cost acclivity assumption

Specific to this work, we make an assumption on the behaviour of PIC which forms the basis of analysis in Section 3.1. For ease of exposition, we will first explain the assumption for 2D `ESS`.

2D `ESS` ***case :*** Let X, Y be the two error prone dimensions. Let $q_i = (x_i, y_i)$ be a point in the `ESS`. Let $OC(q)$ be the cost of the optimal plan at selectivity location q. Let $(a, b).q_i$ denote a location q_j where $q_j = (a * x_i, b * y_i)$.

For any fixed $\alpha > 1$, `ESS` satisfies the *cost acclivity assumption* when either of the conditions hold:

1. $OC((\alpha, 1).q_i) \leq \alpha * OC(q_i) \;\; \forall q_i \in$ `ESS`

2. $OC((1, \alpha).q_i) \leq \alpha * OC(q_i) \;\; \forall q_i \in$ `ESS`

Intutively, it implies that when we increase selectivity of one of the dimension by α, then the increase in the optimal cost is atmost α times.

Multi-D ESS ***case :*** In the case of D dimensional ESS, for any fixed $\alpha > 1$, the *cost acclivity assumption* is satisfied when the below condition holds:
If selectivity value of any of the (D-1) dimensions is multiplied by a factor of α *then the optimal cost increases by atmost* α^{D-1} *times.*

3.1 Proposed Solution

[This work has been jointly done with Srinivas Karthik.]

In this section, we will describe a modified version of `PlanBouquet` , where we use the cost acclivity assumption to bound the number of executions per contour. We will start with explaining the notations.

Notations Let $\mathcal{P}$ denote the `PlanBouquet` algorithm, given in [1], with parameters r = 2, and $\lambda = 0.2$.
Let $\mathcal{P}$' denote the modified `PlanBouquet`. Let α be the parameter for which the cost acclivity assumption holds true.

We will first introduce the notion of "Covering Set ", hereafter denoted as CS . CS is a set of selectivity locations such that, for every location q on contour IC, we have one location l in CS which is in the first quadrant of q and cost of the optimal plan at l is atmost α times the cost of the optimal plan at q. Moreover, the locations in the CS_i are positioned such a way that, execution of all the plans in it will ensure that all the locations below the contour IC_i are in the third quadrant of the CS location. We will delay the discussion about algorithm for finding CS to Section 3.1.1.

In comparison with `PlanBouquet`, we incorporate two major changes in $\mathcal{P}'$, which are as follows. In the preprocessing phase, we find out covering set corresponding to each isocost contour in the ESS. In the execution phase, we execute the optimal plans in covering set with an increased cost budget. Assuming that we have finished the preprocessing phase and obtained each CS , we will now discuss the execution phase of $\mathcal{P}'$ for 2D ESS, and then extend to higher dimensions.

2D ESS case. In `PlanBouquet` for 2D ESS, we execute *all plans* on the isocost contour, which ensures that every location on the contour gets *third-quadrant* coverage. Hence the MSO linearly increases with number of plans on the densest contour. In $\mathcal{P}$', rather than executing all the plans on the contour, we execute plans in covering set which will cover all the locations below the contour.

Our idea is clearly depicted in Figure 3.1, which shows a contour in 2D ESS. Plans $P1 \ldots P14$ are the plans on the iso-cost contour(IC_i), and plans P_a, P_b, P_c, P_d, P_e are the plans in the covering set (CS_i). We assume that (ϵ,ϵ) is the minimum selectivity possible in the ESS. y_{min} denotes the minimum selectivity on dimension y for a contour. Let $q = (x, y_{min})$ be a location on the contour IC_i. Let location $q_1 = (1, \alpha).q$. Due to the cost acclivity property, we know that $OC(q_1) \leq \alpha * OC(q)$. We can see from Figure 3.1, that plan P_e is the optimal plan at q_1, hence, by executing P_e with cost budget of $\alpha * \texttt{CC}(IC_i)$, we can prune all the locations on the contour IC_i between line $y = y_{min}$ and $y = \alpha * y_{min}$. Similarly, execution of all the plans in CS_i will prune all the locations below the contour. Similar to PlanBouquet, if none of the plans in CS_i complete the execution, then we jump to CS_{i+1} until we reach the maximum selectivity. The case when the actual selectivity of one of the epp is learnt, we stay on the same contour with one predicate less in epp and the corresponding lower dimensional ESS.

Extending to higher dimensions. For a D dimensional ESS, the algorithm to find the covering set is explained in Algorithm 2 which would be discussed later. The unique characteristic about CS_i in higher dimensions is that, the cost of the optimal plan at these locations is atmost $\alpha^{D-1} * \texttt{CC}(IC_i)$, hence we execute the plans in each CS_i with an increased cost budget of $\alpha^{D-1} * \texttt{CC}(IC_i)$ to prune all the locations below the contour.

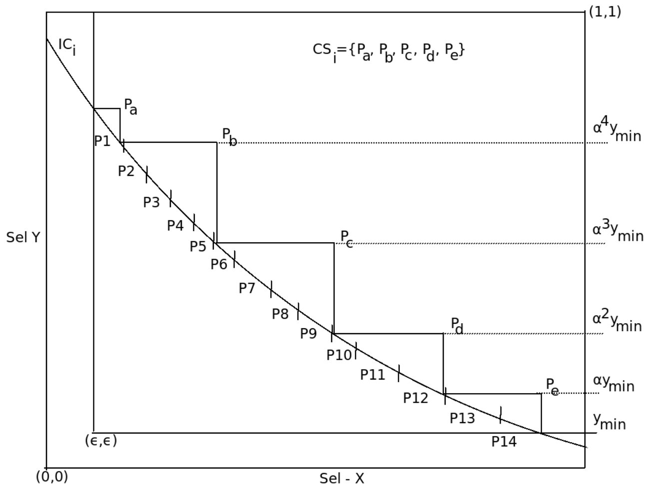

Figure 3.1: Covering Set

Now, in the next section, we will discuss the preprocessing step which focuses on the algorithm to find the covering set . First, we will explain for 2D ESS and then generalize the idea for higher dimensions.

3.1.1 Finding the covering set

For 2D ESS: Algorithm 1 finds all the plans in each covering set for a 2D ESS. Let us first see the notations used in this section.

Notations: $\texttt{PL}(CS_i)$ is a list of plans contained in CS_i. Let P_q denote the optimal plan at the location q. $OC(q)$ denotes the cost of the optimal plan at the location q in the ESS. Let y_{min} denote the minimum selectivity on dimension Y for a contour.

Algorithm 1, starts with the line $y = y_{min}$. Using binary search technique, it then finds the isocost contour location (x_{cur}, y_{cur}) on this line. Then, we scale the y-coordinate α times to obtain the first location $(x_{cur}, \alpha * y_{cur})$ for CS_i. Line 7 in the algorithm finds appropriate x_{cur} so that optimal plan at $(x_{cur}, \alpha y_{cur})$ gives third-quadrant coverage for all locations on IC_i between $[y_{cur}, \alpha y_{cur}]$. In each iteration, we jump to next line by multiplying y_{cur} by α. The update of y_{cur} in line 10, ensures that the **while** loop terminates in atmost $\log_\alpha \frac{1}{\epsilon}$ steps, which would also serve as the upper bound on $|\texttt{PL}(CS_i)|$. It should be noted that all these plans are executed with same budget of $\alpha * \texttt{CC}(IC_i)$.

Input : $\alpha > 1$,x_{min},y_{min}
Output: $\texttt{PL}(CS_i)$ - Plans in the i^{th} covering set
1 Initializations: $i = 1$, $\texttt{PL}(CS_i) = \emptyset$,
; `//` i `denotes the current contour`
; `//` $\texttt{PL}(CS_i)$ `denotes the set of execution plans in covering set` CS_i
2 **repeat**
3 Let y_{min} be the minimum y-coordinate value in IC_i;
4 $y_{cur} = y_{min}$;
5 **while** $y_{cur} \leq 1$ **do**
6 Find x_{cur} :
7 Do a binary Search on line y = y_{cur}
8 from x = x_{min} to x = 1 such that
9 $OC(q_{cur}) = \texttt{CC}(IC_i)$, where $q_{cur} = (x_{cur}, y_{cur})$;
10 $y_{cur} \leftarrow y_{cur} * \alpha$;
11 **if** $y_{cur} > 1$ **then**
12 $y_{cur} = 1$;
13 **end**
14 Let $q = (x, y_{cur})$;
15 **if** *Plan* P_q *is not in* $\texttt{PL}(CS_i)$ **then**
16 Add P_q to $\texttt{PL}(CS_i)$;
17 **end**
18 **end**
19 i = i + 1;
20 **until** *All the contours are visited*;

Algorithm 1: Find covering set for IC_i in a 2D ESS

Multi-D ESS Case: In Algorithm 2, we have extended the procedure of finding plans in each CS_i for D dimensional ESS. The basic idea is to reduce dimensions one by one by intersecting hyperplanes of one dimension less.

For ease of exposition, let us take an example of 3D ESS with X, Y, Z being the three selectivity dimensions. Now, each of the iso-cost contours IC_i is a 3D surface. Let us assume that the cost acclivity assumption holds true for the dimensions Y,Z with α as the parameter. We first intersect hyperplane $z = z_{min}$ with the IC_i surface, and we get a 2D contour with selectivity dimensions as X and Y. Let $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ be the locations in covering set corresponding to the 2D contour obtained above with $z = z_{min}$. So the plans on the locations $(x_1, y_1, \alpha z_{min}), (x_2, y_2, \alpha z_{min}), \ldots, (x_n, y_n, \alpha z_{min})$ will cover all the locations on IC_i from z=z_{min} to z=αz_{min}. Similarly, we can find the plans for $z = \alpha z_{min}$ and so on. Due to the cost acclivity assumption, if we execute plans in covering set CS_i with cost budget of $\alpha^2 * \texttt{CC}(IC_i)$, then all the locations below the contour IC_i would be pruned.

In D dimensional ESS , all the plans in $\texttt{PL}(CS_i)$ are executed with the budget $\alpha^{D-1} * \texttt{CC}(IC_i)$.

3.2 Theoretical Results

3.2.1 Preprocessing time incurred

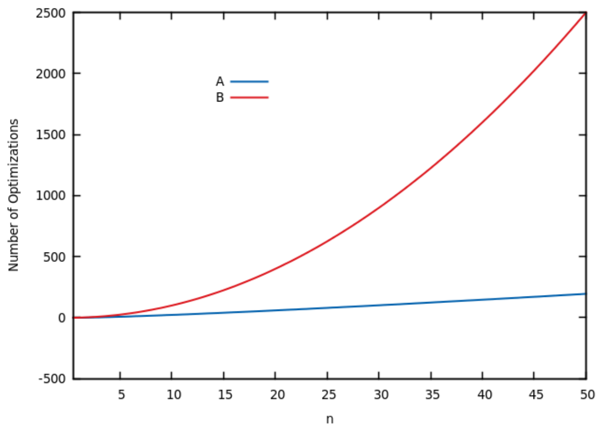

Figure 3.2: Comparison of the number of optimizations

Let us see the theoretical bound for the preprocessing time incurred for our algorithm. In algorithm 2, Line 8 takes $log(\texttt{res})$ number of optimizations to perform the binary search, and **for** loops from line 3 to line 6 iterates $\log_{\alpha}^{D-1} \frac{1}{\epsilon}$ times. So total number of optimizations in our

Input : $\alpha > 1, \forall i \;\; x_i^{min}$

Output: $\texttt{PL}(CS_i)$ - Plans in the i^{th} covering set

```
1  Initializations: PL(CS_i) = ∅, i = 1;
2  repeat
3      for x_1 ← α * x_1^min to 1 do
4          compute x_2^min at x_1/α ;
           ⋮
5          compute x_{D-1}^min at (x_1/α, ··· , x_{D-2}/α);
6          for x_{D-1} ← α * x_{D-1}^min to 1 do
7              Find x_D :
8                 Do a binary Search
9                 from x_D = x_D^min to x_D = 1 such that
10                OC(q_D) = CC(IC_i), where q_D = (x_1/α, ··· , x_{D-1}/α, x_D);
11             Let q = (x_1, x_2, ··· , x_D);
12             if Plan P_q is not in PL(CS_i) then
13                 Add P_q to PL(CS_i);
14             end
15             x_{D-1} ← x_{D-1} * α;
16         end
           ⋮
17         x_1 ← x_1 * α;
18     end
19     i = i + 1;
20 until All the contours are visited;
```

Algorithm 2: Find covering set for IC_i in Multi-D $\texttt{ESS}$

approach is given by

$$log(\texttt{res}) * \log_{\alpha}^{\texttt{D-1}}(\frac{1}{\epsilon})$$

The comparison between the functions is shown in Figure 3.2. For $\epsilon = 0.01$, $\alpha = 1.6$ and $d = 2$, B denotes the function res^D and A denotes the function $log(\texttt{res}) * \log_{\alpha}^{\texttt{D-1}}(\frac{1}{\epsilon})$.

3.2.2 Impact on MSO Bound

Now, we will try to formulate the MSO Bound for $\mathcal{P}'$. During the execution time of algorithm $\mathcal{P}'$, we execute the plans in the covering set (CS_i) with budget $\alpha^{D-1} * \texttt{CC}(IC_i)$ for each plan contained in CS_i. If execution of none of the plans in CS_i gets completed, then we jump to covering set CS_{i+1}. In Algorithm 2, due to the update in line 14, we can ensure that the total number of iterations for a single contour can be utmost $\log_{\alpha}^{D-1}(\frac{1}{\epsilon})$. Suppose that the actual selectivity location q_a is located in the range $(IC_k, IC_{k+1}]$. Then, the algorithm explores the contours from 1 to $k+1$ before discovering q_a. In the following lemma we show that, by using $\mathcal{P}'$, we get MSO bound independent of the plan behaviour (i.e ρ)

Lemma 3.1. *$MSO \leq 4\alpha^{D-1}\log_{\alpha}^{D-1}(\frac{1}{\epsilon})$.*

Proof.

$$\begin{aligned}
\text{Total Cost} &= \sum_{i=1}^{k+1} \#\text{ plans per contour*Budget} \\
&\leq \log_{\alpha}^{D-1}(\frac{1}{\epsilon})(\sum_{i=1}^{k+1}\alpha^{D-1} * \texttt{CC}(IC_i)) \\
&= \log_{\alpha}^{D-1}(\frac{1}{\epsilon}) * (\alpha^{D-1}) * (\sum_{i=1}^{k+1}\texttt{CC}(IC_i)) \\
&= 4\alpha^{D-1}\log_{\alpha}^{D-1}(\frac{1}{\epsilon}) * \texttt{CC}(IC_k)
\end{aligned} \tag{3.1}$$

The cost for an oracle algorithm that apriori knows the correct location of q_a is lower bounded by $\texttt{CC}(IC_k)$. Hence,

$$\therefore MSO \leq 4\alpha^{D-1}\log_{\alpha}^{D-1}(\frac{1}{\epsilon}) \tag{3.2}$$

Thus the above MSO expression is independent of the plan density (ρ) for each contour. For clarification of the reader, we stress on the fact that, the above MSO expression cannot be degenerated to an expression with ρ. This is because, for a fixed ϵ, α and D, we will get a fixed value for the MSO bound, using the above expression, whereas ρ will depend upon the complexity of the optimizer. □

3.3 Experimental Evaluation

In this section, we will give the empirical results tested on TPCH [5] benchmark queries. The database engine used is PostgreSQL 9.3.4 [6]. We use the standard 1GB TPCH database. As in the paper [1], the physical schema has indexes on all columns featuring in the queries, thereby maximizing $\frac{C_{max}}{C_{min}}$, creating "hard-nut" environments for achieving robustness, where C_{max} is the cost of optimal plan at highest selectivity on all error-prone dimension and C_{min} is the cost of the optimal plan at the lowest selectivity on all the error-prone dimensions.

We will first validate the cost acclivity assumption. We checked for most of the TPCH queries with 2D and 3D ESS and it was found that almost 99% of the points satisfies the cost acclivity assumption with $\alpha \in [1.6, 2]$. In Table 3.1, we have shown the percent violation for different query templates.

Now, we compare the reduction in the overall pre-processing time after using $\mathcal{P}'$. We obtained around 98% reduction in the preprocessing time, when compared with the naive preprocessing step mentioned in `PlanBouquet`.

Next, we compare the query template specific MSO Bounds for $\mathcal{P}$ and $\mathcal{P}'$. We notice that, even though we removed the dependency of ρ from the MSO expression, the MSO bound, specific to a query for $\mathcal{P}'$ is worse than $\mathcal{P}$. Table 3.2 shows the MSO Bound for 5 query templates using $\mathcal{P}'$ with $\epsilon = 4.7 * 10^{-6}$. Here α value chosen is 1.6. The value of ϵ used, is obtained as the lowest selectivity point using the exponential distribution with resolution 300. Since `PlanBouquet` technique uses the anorexic reduction technique, the MSO Bound for any given query template is around 40, which is considerably low as compared to the numbers in Table 3.2. Alternatively, we could use ϵ value as the selectivity at which we get one tuple from the table on which we have the error-prone predicate.

Finally, Table 3.3 shows the empirical MSO obtained for query template 5 with 2D `ESS`. In the table, second column PB(FPC), denote the MSO obtained using `PlanBouquet` with anorexic reduction. The result without using reduction is shown in third column. The result obtained for our approach is shown in the last column. We used the same values of ϵ and α as earlier experiment. We can see that the MSO values for our approach are considerably higher than both the variants of `PlanBouquet`. The selectivity location where the worst case impact happens, are different for both of them.

We conclude by saying that this initial idea did not provide a substantial empirical benefit, rather, it gave a theoretical perspective to look for improvements in `PlanBouquet` by harnessing the properties of the cost functions.

Having discussed this initial idea for `PlanBouquet`, we will now see our second contribution, where we tried to improve an intrusive system. In the next chapter, we will see the magnitude of improvement which we can achieve by using the power of intrusiveness into the database engine.

		Percent violation of $\mathcal{A}$	
QT	α	2D	3D
QT2	1.6	1.32	0.77
	1.7	1.11	0.82
	1.8	1.02	0.91
	1.9	0.74	1.18
	2	0.65	0.39
QT5	1.6	1.1	0.86
	1.7	0.93	0.96
	1.8	0.81	0.40
	1.9	0.71	0.11
	2	0.31	0.20
QT7	1.6	0.96	0.41
	1.7	0.73	0.02
	1.8	0.35	0.01
	1.9	0.21	0.01
	2	0.11	0.01

Table 3.1: Percent violation of *cost acclivity assumption*

QT	$\log_{\alpha}^{D-1}(\frac{1}{\epsilon})$	MSO Bound
2	7	44.77
5	11	74.80
7	9	64.79
8	11	83.59
9	5	40.00

Table 3.2: MSO Bounds of $\mathcal{P}$' for 2D ESS

QT	PB(FPC)	PB(w/o FPC)	Our Approach
QT5	12.61	14.32	19.15

Table 3.3: Empirical MSO

Chapter 4

Improving intrusive technique

In the initial work, we looked at PlanBouquet, a non-intrusive robust query processing approach and improved the MSO bound expression to be independent of ρ, by harnessing a property of PIC. But the bound obtained, in Section 3.2.2 contained a α^{D-1} term, which can be high for higher dimensions.
Now, we switch gears and try to tweak into the database engine, to achieve better worst case guarantees. We will now discuss the improvements which we can get, when we are given power of intrusiveness into the database engine. In our lab, Srinivas Karthik has come up with an intrusive technique called SpillBound[2], which changes the execution component to achieve better overall performance. In this thesis, we propose Opt-SB, which tries to improve the MSO bound given by SpillBound. Before we get into our work, we will briefly explain SpillBound.

4.1 SpillBound Algorithm [2]

PlanBouquet algorithm, as summarized in Section 2.4, *implicitly* discovers selectivities through the completion status of cost-budgeted plan executions whereas SpillBound[2] tries to improve the MSO bound by *explicitly* monitoring and accelerating the discovery process. It leverages the notion of "spilling" which involves modifying the execution of a plan to extract increased learning about selectivities within the assigned execution budget.

A location $q \in$ ESS is said to be "below" contour IC_i, if there exists a location $q_c \in IC_i$ such that $q \preceq q_c$. On the other hand, a location $q \in$ ESS is "above" IC_i, if there is a location $q_c \in IC_i$ such that $q \succ q_c$. The notations also apply to a region, consisting of contiguous locations, with respect to a contour.

Figure 4.1 illustrates the idea of spilling. Consider the plans P1 and P2. S, L, O, C, N denote tables from TPC-H benchmark which are namely Supplier, Lineitem, Orders, Customer

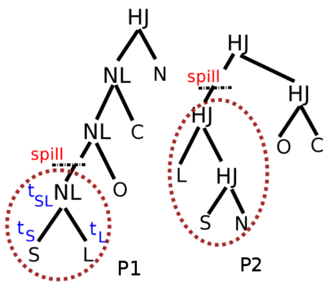

Figure 4.1: Spilled Plans

and Nation respectively. NL and HJ stands for Nested-loop join and Hash join respectively. Assume that the S-L join predicate is erroneous in this case. The execution of plans P1 and P2 with a cost-budget of c, takes place in a bottom up fashion, i.e the tuples are moved from the leaf nodes to the root of the plan tree. For an internal node, the set of nodes which are in the subtree rooted at the node is called as its *upstream* nodes, and the set of nodes on its path to the root as its *downstream* nodes. The cost of the whole plan is obtained by summating the cost of all the internal nodes. Since `SpillBound` interested in finding the selectivity of the erroneous node S-L, the part of the budget that is assigned for its downstream operators is not useful for learning about its selectivity. So, spilling out the results of the S-L node, without forwarding to downstream nodes helps to use the budget more effectively to learn about its selectivity. This idea of spilling helps to achieve a lower bound on the selectivities of error-prone predicates, for instance S-L join predicate. The node used for spilling is termed as *spill node*.

A formal procedure for identifying the spill node is given in [2]. The identification of spill node ensures that the selectivities of all the predicates that are upstream of the chosen spill node are exactly known. A desirable property for the spill node identification procedure is to carefully choose one of the `epp` which gives a guaranteed selectivity learning. For instance, while exploring a location $q \in$ `ESS`, when q does *not* dominate q_a, in the spilling approach, it learns that $q_a.j > q.j$, where R_j is the spill node identified for the optimal plan at q. Plan P_i is said to spill on dimension k, if `epp` used as a spill node for plan P_i corresponds to dimension k. Given an execution plan, an identified `epp` to spill on and a cost budget B, the modification to the plan to enable spilling is presented in Algorithm 3.

The difference between the plan bouquet approach and the spilling approach is pictorially shown in Figure 4.2, for the region of interest below contour IC_i. In plan bouquet approach, when plan P_3 is executed, it will prune only the blue region of the `ESS`. Whereas in `SpillBound`, if P_3 has spill node corresponding to dimension X, then a spill-mode execution of P_3, would *additionally* prune green region. Similarly if P_3 spills on Y, then it will prune the yellow and

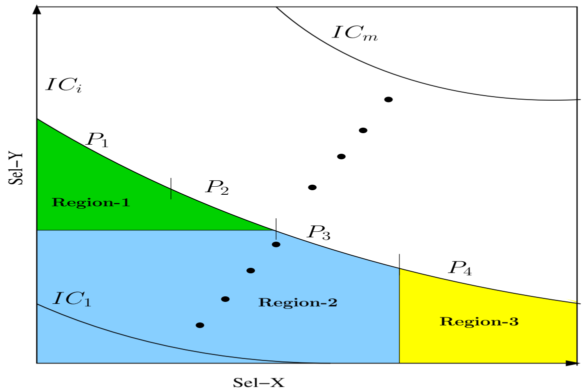

Figure 4.2: Pruning of ESS (`PlanBouquet` and `SpillBound`)

the blue region.

Algorithm 3: Spill-Mode-Execution
Input: Execution plan P, it's spilling predicate `epp` R_j, and cost budget B; Create a *modified-plan* which is the subplan of P rooted at R_j; Execute the modified-plan until the budget B is exhausted; **Output:** Selectivity learnt for R_j after execution;

Now, having discussed the concept of spilling, we will briefly explain the `SpillBound` algorithm. To understand this algorithm, let us first start with a case of 2D `ESS` and then generalize to the multi-dimensional case.

4.1.1 The `2D-SpillBound` Algorithm

Let the two `epp` dimensions be denoted by X and Y. In Figure 4.2, if P_2 and P_3 both spill on dimension X, then plan P_2's execution can be avoided since the guaranteed learning/lower bound on selectivity of X from P_3 subsumes the learning obtained from P_2. Hence just two executions are enough to prune the entire region below the contour if the q_a lies outside the pruned region.

They annotate each plan with the predicate it can spill on. For instance, P_0^x denotes that plan P_0 is identified to spill on predicate X. Further, `SpillBound` are interested in two plans $P_i^{x_{max}}$ and $P_i^{y_{max}}$ in each contour IC_i, one which maximizes the lower bound on learning selectivity of X and other for Y. Formally, $P_i^{x_{max}}$ will be the optimal POSP plan at location $q_i^x \in IC_i$, where

$$q_i^x = \underset{q}{\text{argmax}}\{q.x | P_q \text{ is identified to spill on } \texttt{epp } X\}$$

For the axes specified in Figure 4.2, $P_i^{x_{max}}$ denotes the rightmost location plan in the contour which can spill on X. The plan $P_i^{y_{max}}$, which is defined similarly, would be the top most location plan in the contour which can spill on Y.

The algorithm explores the ESS contour-wise, from the minimum cost contour, while executing two plans $P_i^{x_{max}}$ and $P_i^{y_{max}}$ in spilling mode sequentially which were chosen using the spill node identification process. This contour-wise exploration continues until one of the epp learns its actual selectivity, say epp X (or equivalently, $q_{run}.x = q_a.x$). Then it knows that q_a lies on the line $x = q_{run}.x$. After this, the standard PlanBouquet algorithm is used to discover the selectivity of the remaining predicate, starting from the present contour while executing plans in non-spilling mode. Since the problem is reduced to single dimension, only one plan is executed in each contour. The steps of choosing of plans $P_i^{x_{max}}$ and $P_i^{y_{max}}$, and executing them, is carried out only when the corresponding plans exist.

4.1.2 Extending to higher dimensions

Let the number of error-prone predicates be denoted by D, which induces a D-dimensional error-prone selectivity space. The key idea remains the same as in the two dimensional case, wherein by carefully choosing and executing just D plans, the whole region below a contour is pruned. In essence, these plans together provide the maximal learning for all the dimensions, and hence eschew the need of executing the rest of the plans in the contour.

Notations : Let EPPs $= \{R_1, \ldots, R_D\}$ denote the set of error-prone predicates. for any contour IC_i. Let $P_i^{j_{max}}$ denote the plan which can spill on predicate R_j while providing the highest lower bound on selectivity learning in the contour (analogous to plans $P_i^{x_{max}}$ and $P_i^{y_{max}}$, in two dimensional case). Formally, $P_i^{j_{max}}$ is the POSP plan at location $q_i^j \in IC_i$ where

$$q_i^j = \underset{q}{\operatorname{argmax}}\{q.j | P_q \text{ is identified to spill on } \texttt{epp } R_j\} \tag{4.1}$$

As in two dimension case, SpillBound algorithm explores the ESS contour-wise, from the minimum cost contour. At a *generic* step in the algorithm, the spill node identification process is used to identify the set of plans $\{P_i^{j_{max}}\}$ for each epp R_j. This identified set of plans are then executed in spilling-mode, sequentially. As a consequence, q_{run} is updated based on the selectivities learnt in each execution. This wonted sequence of execution is halted when one of the epp learns its actual selectivity. Wherein the *generic* step procedure is repeated while staying in the same contour, but with one less predicate in epp and corresponding lower dimensional ESS. They call this as a *Repeat Step*. They need to revisit the same contour again because it ensures that for all epp dimensions whose actual selectivity location lie below the contour IC_i, are indeed discovered on contour IC_i before moving to the next contour.

The worst case execution occurs when all the repeat steps happen at the last contour, in which case MSO is given by the following theorem :

Theorem 4.1. *[2] The MSO of the* `SpillBound` *algorithm for any query with D error-prone predicates is bounded by $D^2 + 3D$.*

4.2 Proposed Solution

`SpillBound` algorithm, as described in previous section, executes D plans in a D-dimensional `ESS`, and gives a MSO bound of $D^2 + 3D$. We observe that, in the worst case scenario, we will need to execute the D plans to ensure that all the selectivity locations under the contour are discovered. But in practical scenarios, if we carefully replace some plans on each contour, we may not need to execute all the D plans per contour. This is the high-level idea for our algorithm `Opt-SB`. We will now explain `Opt-SB`. Firstly, we will briefly describe the notations used in this work, and then move ahead to explain the idea.

Notations : For a given contour IC_i, location q is said to be an *extreme location on dimension i* if q has the maximum selectivity on dimension i amongst all the locations on the contour. For instance, in Figure 4.3, q_1^x is an extreme location on dimension X for contour IC_1.

A contour IC_i is said to be *aligned on dimension i*, if the extreme location on dimension i, q ($q \in IC_i$) has an optimal plan which spills on dimension i. For example, in Figure 4.3, if query location q_2^y has a spill node as the `epp` corresponding to dimension Y, then we say that contour IC_2 is aligned on dimension Y.

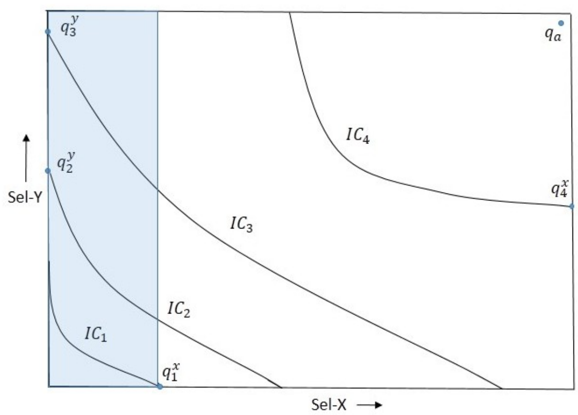

Figure 4.3: Favourable case for `SpillBound`

4.2.1 Favourable Case

We will start by discussing a favourable case for `SpillBound`. *Favourable case occurs for* `SpillBound` *when all the contours are aligned on at least one of the dimensions.* For example, consider the 2D ESS as shown in Figure 4.3. Here, contours are denoted by IC_1, IC_2, IC_3, IC_4. For each contour IC_i, we have shown one of the two extreme locations, which is denoted by either q_i^x or q_i^y. Consider that all the contours are aligned on the dimension, on which the extreme location is shown in the figure, i.e, for every location q_i^j shown in Figure 4.3, it has an optimal plan that spills on dimension j.

For q_a, as shown in the figure, let us see how the executions will take place using `SpillBound`. Initially, optimal plan at q_1^x would be executed in spill mode with the budget $\texttt{CC}(IC_1)$, which does not finish its execution. So, due to the property of spilling, we will get a guaranteed selectivity learning for `epp` R_x, which is $q_a \cdot x > q_1^x \cdot x$. Since q_1^x is the location on IC_1 with maximum selectivity for dimension X, we prune all the selectivity locations ($q \in \texttt{ESS}$) where $q \cdot x \leq q_1^x \cdot x$. The shaded region in Figure 4.3 represents the pruned space of the `ESS` when optimal plan at q_1^x is executed in spill mode. Since the shaded region encompasses the whole contour, there is no need of another execution in the same contour. Hence, only one execution is needed per contour in this case.

Let us now see the case when we learn selectivity on one of the `epp` completely. Here, we need to revisit the contour with one less `epp` and the corresponding lower dimensional `ESS`. Let us denote step of revisiting a contour as a *Repeat Step.* So, for D dimensions, if all the contours are aligned on some dimension, we will need only one execution per contour in addition to the *Repeat Steps.* Moreover, after each repeat step, the number of dimensions of `ESS` reduces by exactly one. Thus, at most there could be D repeat steps in the entire execution sequence. With the above explanation, we will now give the MSO bound for the favourable case.

We will be proving the bound for a general D dimensional ESS. Let us assume, without loss of generality, that the actual selectivity location q_a is located in the range $(IC_k, IC_{k+1}]$. Then, the algorithm will explore the contours from 1 to $k + 1$ before discovering q_a. In the worst case execution sequence, we will encounter all the D repeat steps at the last contour (k+1). So the total cost incurred would be given by

$$
\begin{aligned}
TotalCost &= \texttt{CC}(IC_1) + \cdots + \texttt{CC}(IC_k) + D * \texttt{CC}(IC_{k+1}) \\
&= \texttt{CC}(IC_1) + \cdots + 2^{k-1}\texttt{CC}(IC_1) + D * 2^k\texttt{CC}(IC_1) \\
&= \texttt{CC}(IC_1) * (1 + 2 + \cdots + 2^{k-1}) + D * 2^k\texttt{CC}(IC_1) \\
&= (2^k - 1)\texttt{CC}(IC_1) + D * 2^k\texttt{CC}(IC_1) \\
&\leq (2^{k-1}\texttt{CC}(IC_1))(2D + 2)
\end{aligned}
$$

From the plan cost monotonicity (PCM) assumption, we know that the cost for an oracle algorithm that apriori knows the correct location of q_a is lower bounded by $\texttt{CC}(IC_k)$. By definition, $\texttt{CC}(IC_k) = 2^{k-1} * \texttt{CC}(IC_1)$. Hence,

$$
\begin{aligned}
\therefore MSO &\leq \frac{TotalCost}{2^{k-1} * \texttt{CC}(IC_1)} \\
&\leq 2D + 2
\end{aligned}
$$

$$
\boxed{MSO \leq 2D + 2} \tag{4.2}
$$

Thus, for the favourable case of `SpillBound`, we get a MSO bound that is linear in the number of dimensions.

Practically, we found out that not all contours are aligned for a given ESS. So, we try to align each contour by replacing the optimal plan at the extreme locations of misaligned contours with a plan from POSP that spills on the same dimension. Further, the replacement comes at a cost, which is the cost of forcing the replaced plan at the extreme location. For experiments, plan replacement was carried out using the feature of "Foreign Plan Costing"[7]. We will denote the percentage increase in the cost incurred while replacing the plan as *tolerance* t.

Now, for spilling to work, we need to execute these plans with a budget equal to their cost at the forced location, hence we increase the cost-budget for each contour by t percentage. Since we have increased the cost budget for a contour, this will have an effect on the MSO bound. If MSO_{orig} was the original MSO bound and MSO_{new} is the MSO bound after increasing the cost budget with tolerance t, then the following holds true.

$MSO_{new} = (1 + \frac{t}{100})MSO_{orig}$

Let us empirically see, the minimum tolerance required to align all the contours for an `ESS`. Empirically observed minimum tolerance is shown in Table 4.1. We have carried out experiments on queries from TPC-H and TPC-DS benchmarks with their standard sizes of 1 GB and 100 GB respectively. The nomenclature for the queries is xD_y_Qz, where x specifies the number of error-prone dimensions, y the benchmark (H or DS), and z the query number in the benchmark. So, for example, 3D_H_Q5 indicates a three-dimensional error-prone selectivity space on Query 5 of the TPC-H benchmark.

Query Template	Tolerance(%)	Our MSO	SpillBound MSO
2D_DS_Q96	66	10	10
3D_DS_Q15	25	10	18
3D_DS_Q96	12251	988	18
4D_DS_Q7	276	31	28
4D_DS_Q91	339	43	28
4D_DS_Q26	5047	514	28
5D_DS_Q19	5	13	40
3D_H_Q5	0	8	18
3D_H_Q7	12	9	18
4D_H_Q8	1	10	28
5D_H_Q7	1.6	12	40

Table 4.1: Minimum tolerance needed to align all contours.

For some of the query templates mentioned in the above table, the tolerance required is more than 200%. Specifically, for TPC-DS query template 96 with 3 error prone dimensions, is as high as 12251, which increases the MSO bound given by `SpillBound` to 988. So we observe that, this approach cannot be used in practice. Figure 4.4, gives a broader perspective to analyze `SpillBound`. Number of executions in favorable case is 1, whereas in worst-case, it goes up to D. We try to improve `SpillBound`, by analyzing the current situation, and see if the number of executions per contour could be between 1 and D. We design a new algorithm named `Opt-SB` which will dynamically decide the number of executions required per contour. The property of this algorithm is that, in best case, it will give a $O(D)$ bound, while in the worst case, it gives a bound of $O(D^2)$. Let us now see how `Opt-SB` works.

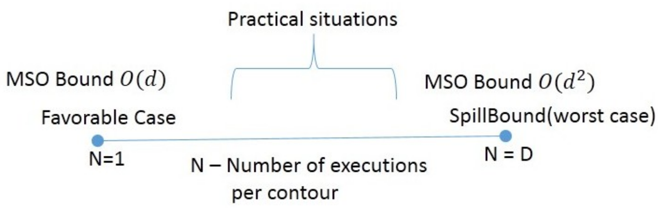

Figure 4.4: Big Picture

4.2.2 `Opt-SB`

As we have seen in the previous section, that tolerance to align each contour is huge. So, rather than aligning each contour to achieve one execution per contour, we can align *partitions of contour* and achieve a favorable case for each of these partitions. We call these partitions of contours as *subcontours*, which will be precisely defined later in this section. So, by executing one plan per subcontour, we can cover the whole contour. So, the number of executions per contour would be equal to the number of sub contours. This is the central idea behind `Opt-SB`. Before going into details of the algorithm, let us see the notations used.

Notations: D is the number of dimensions of `ESS` where we number each dimension starting from 1.
Let p denote the number of partitions. Let $T_1, \ldots, T_p$ denote the partitions of the D dimensions. For example, in a 4D ESS, let $T_1 = \{1, 2\}, T_2 = \{3, 4\}$ represent two partitions, where dimensions 1 and 2 belong to partition 1 and dimensions 3 and 4 belong to partition 2. For ease of exposition, we redefine two notations which is as follows:
Let K be a subset of selectivity locations of `ESS`. Location q_i ($q_i \in K$) is said to be an *extreme location of K on dimension i* if q_i has the maximum selectivity on dimension i amongst all the locations in K. We say that K is *aligned on dimension i with tolerance t*, if there exist a plan A^i in POSP, which spills on dimension i, such that $Cost(A^i, q_i) \leq (1 + \frac{t}{100}) * OC(q_i)$. Any such plan, A^i with tolerance t, is called as a *leader plan* of K, denoted as LP(K) and dimension i is termed as a *leader dimension*, denoted as LD(K). It could be possible that there are more than one leader plan or leader dimension for K, but for simplicity, we will randomly consider any one of the many choices for LP(K) and LD(K).

Now let us try to understand `Opt-SB` as given in Algorithm 4. For a given query, `Opt-SB` uses a preprocessing module to obtain the partitions $T_1, \ldots, T_p$ and the maximum tolerance t. We will delay the discussion of preprocessing module until we prove the MSO expression.

The algorithm explores the `ESS` contour-wise, from the minimum cost contour. After dividing D dimensions into p partitions, $T_1, \ldots, T_p$, we run the spill node identification procedure for all the locations in IC_i. Now, we will introduce the notion of a subcontour. After we have partitioned the dimensions into p partitions, we split the contour locations of each IC_i, into p sets, $S_1, \ldots, S_p$, where S_i is a set of all the locations on the contour IC_i whose optimal plan spills on one of the dimensions included in T_i. For example, in the above 4D `ESS`, S_1 will contain all the locations on the contour which has optimal plan that spills on either dimension 1 or 2. We term each of these S_i as a *subcontour*. The preprocessing module ensures that we can align each subcontour with tolerance of t. After aligning each subcontour in IC_i, we denote the leader plan and the leader dimension for each of these subcontours as P_j and d_j respectively. We perform a Spill-Mode-Execution (Algorithm 3) of all the leader plans along with corresponding leader dimension with an increased cost budget of $\texttt{CC}(IC_i) * (1 + \frac{t}{100})$. In line 16, Sel-Learnt denotes the selectivity of dimension d_k, which was learnt after executing P_k in spill mode. We use this learnt selectivity to update q_{run} accordingly. This sequence of p-executions-per-contour is halted when one of the `epp` learns its actual selectivity. Wherein the *generic* step procedure is repeated while staying in the same contour, but with one less predicate in `epp` and corresponding lower dimensional `ESS`. We also remove the learnt dimension from its respective partition. The intuitive rationale behind revisiting the same contour, guided by the boolean variable Repeat-Contour, is to prune the region below the contour. In other words, it ensures that all the selectivity locations lying below the contour IC_i are indeed discovered before moving to the next contour, which is critical for the required MSO bound in the end.

Since all the subcontours are aligned with tolerance t, only one plan execution is needed for each subcontour to cover all the locations below that subcontour. Thus, there would be at most p executions per contour to prune the entire region below the contour, if q_a lies outside the pruned region.

Let us denote the current number of partitions by p_{cur}. We use (IC_i, R_j) pair to identify the executions which spill on `epp` R_j in contour IC_i. As the name suggests, we call the first execution of the (IC_i, R_j) combination as a *fresh step*, and subsequent executions as *repeat steps*. We need to observe that *repeat steps* happen only after Repeat-Contour variable is set to true in each contour. Consider any contour IC_i. Note that the number of possible fresh executions of the form (IC_i, R_j) on contour IC_i for `Opt-SB` is bounded by p (in fact, it is equal to p_{cur}, when the algorithm enters the contour IC_i iteration).

As mentioned earlier, a repeat step can happen only when the variable Repeat-Contour is set to true in any contour. After selectivity of any one of the `epp` is learnt, we remove the dimension corresponding to the `epp` from its respective partition. However, this may not decrease the p_{cur}, since the partition may contain more than one dimension. So, in the worst case, we will have p executions for (D-p) repeat steps, until we reach a state where each partition has only one dimension. After coming to this state, let us say that, when Repeat-Contour is set to true, it marks the beginning of a new phase. It is easy to see that, there would be $p_{cur} - 1$ repeat steps within a phase, where p_{cur} refers to the number of partitions which were present at the beginning of a phase. Further, in each phase, the size of p_{cur} decreases by 1. Therefore, the number of repeat steps is bounded by $\sum_{l=1}^{p-1} l = \frac{p(p-1)}{2}$.

We will now present the theoretical bound on MSO for `Opt-SB`. Suppose that the actual

Algorithm 4: The `Opt-SB` Algorithm

Init: i=1, `EPPs`=$\{R_1, \ldots, R_D\}$;
Initialize(π, t) from the Preprocessing module;
Let p be the number of partitions in π;
Let the corresponding partitions be denoted by $T_1, \ldots, T_p$;
while $i \leq m$ **do** { for each contour}
  if $|$`EPPs`$| = 1$ **then** { only one `epp` left}
    Run the 1D `PlanBouquet` algorithm to discover the selectivity of the remaining `epp` starting from the *present* contour;
    Exit;
  end if
  Run spill node identification procedure on each plan in the contour IC_i
  Partition IC_i into subcontours $S_1, \ldots, S_p$
  Let $P_j = \text{LP}(S_j)$ for $1 \leq j \leq p$
  Let $d_j = \text{LD}(S_j)$ for $1 \leq j \leq p$
  Repeat-Contour = false;
  for each partition k, $1 \leq k \leq p$ **do**
    Sel-Learnt = Spill-Mode-Execution$(P_k, R_{d_k}, \texttt{CC}(IC_i))$;
    Set $q_{run}.d_k$ to Sel-Learnt;
    if $q_{run}.d_k = q_a.d_k$ **then** { learnt its actual selectivity}
      Remove R_{d_k} from partition T_k;
      Remove R_{d_k} from the set `EPPs` ;
      Repeat-Contour = true;
      Break;
    end if
  end for
  if Repeat-Contour = true **then**
    /*Stay on the contour */
  else
    i = i+1; /* Move to next contour */
  end if
  Update `ESS` based on learnt selectivities, i.e. updated q_{run};
end while

selectivity location q_a is located in the range $(IC_k, IC_{k+1}]$. Then, the `Opt-SB` algorithm explores the contours from 1 to $k+1$ before discovering q_a. Recall that $\texttt{CC}(IC_i) = \texttt{CC}(IC_1) \cdot 2^{i-1}$. As we discussed, the worst case execution sequence would be as follows:

- p fresh execution for contours 1 to k+1, without learning the actual selectivity of any `epp`. We will denote this cost as A_1.

$$\therefore A_1 = p * (\texttt{CC}(IC_1) + CC(IC_2) + \cdots + \texttt{CC}(IC_{k+1}))$$
$$\leq 2p * \texttt{CC}(IC_{k+1})$$

- p executions on contour k+1, for learning selectivity of each of the (D-p) dimensions, such that it eventually comes to a state when there is exactly one dimension per partition. Let A_2 denote this cost.

$$\therefore A_2 = 2\texttt{CC}(IC_k) * ((D - p) * p)$$

- Similar to `SpillBound`, $\frac{p(p-1)}{2}$ repeat steps on contour k+1. We will denote this cost as A_3

$$\therefore A_3 = 2\texttt{CC}(IC_k) * (\frac{p * (p-1)}{2})$$

Moreover, since we are using tolerance of t to align each subcontour, the total cost in the worst case execution sequence is bounded as follows:

$$TotalCost \leq (A_1 + A_2 + A_3) * (1 + \frac{t}{100})$$
$$\leq (2pD - p^2 + 3p) * (1 + \frac{t}{100})(\texttt{CC}(IC_k))$$

The cost for an oracle algorithm that apriori knows the correct location of q_a is lower bounded by $\texttt{CC}(IC_k)$. Hence,

$$MSO \leq \frac{TotalCost}{\texttt{CC}(IC_k)}$$

$$\boxed{MSO \leq (2pD - p^2 + 3p) * (1 + \frac{t}{100})} \tag{4.3}$$

We show in Section 4.3 that the tolerance value (t) obtained using Algorithm 4 , in practical situations, is not significantly high.

When the number of partitions (p) is equal to the number of dimensions(D), we degenerate to the case of `SpillBound`. i.e if we substitute $p = D$ in equation (4.3), the MSO expression becomes,

$$\boxed{MSO \leq D^2 + 3D} \tag{4.4}$$

It is easy to see that, when $p = D$, the tolerance value t becomes 0.

Now we will discuss the preprocessing module, as mentioned earlier, which gives the best way to partition the D dimensions. Let π denote a valid partitioning of the D dimensions of ESS into p partitions denoted by $\{T_1 \dots T_p\}$. The total number of possible partitionings for D dimensional ESS is given by the *Bell number*[8]. For each possible partitioning π, we can get the corresponding maximum tolerance t needed to align all the subcontours in the ESS. We exhaustively try out each possible partitioning and choose the optimal one, which minimizes the MSO expression given in equation (4.3). Note that there is an implicit assumption that, when we learn the actual selectivities of some of the EPPs, the maximum tolerance required to align subcontours in the reduced space would be not greater than t corresponding to the current optimal partitioning. As shown in Algorithm 4, we fix the partitioning at the beginning and use it for all the contours. Rather, we could find an optimal partitioning for each contour in the ESS. We call such kind of partitioning as *local-partitioning*, and we have shown results for it in Section 4.3.

This technique of exploring all the possible partitioning, is infeasible when D is high. As a part of future work, we will try to find a heuristic technique which will efficiently search the optimal partitioning in high dimensional ESS.

4.3 Results

In this section, we present an empirical evaluation of Opt-SB on benchmark queries and compare with SpillBound.

4.3.1 Experimental Setup

We have shown results on PostgreSQL version 8.3.6. Since SpillBound is an intrusive technique, we are unable to show results on commercial database engines. We have used Ubuntu 14.04 LTS, on a vanilla Sun Ultra 24 workstation with 8GB of memory and 1TB of hard disk. We have carried out experiments on queries from TPC-H and TPC-DS benchmarks which covers a range of join-graph geometries including chain, star etc. All the error-prone selectivities are chosen as join selectivities providing a challenging multi dimensional ESS. Number of error prone selectivities ranges from 2 to 5. We have used the standard sizes of TPC-H and TPC-DS databases which are 1 GB and 100 GB respectively. The physical schema of the database has index on every column in the database, thus increasing the cost gradient $\frac{C_{max}}{C_{min}}$ and creating a challenging environment for achieving robustness.

Query Template	π_r	t	`Opt-SB`	`SpillBound`
2D_DS_Q96	{1,2}	66	9.9	10
3D_DS_Q15	{1},{2,3}	0	14	18
3D_DS_Q96	{1},{2},{3}	0	18	18
4D_DS_Q7	{1,3,4},{2}	47	26.4	28
4D_DS_Q91	{1,2,3},{4}	4	18.7	28
4D_DS_Q26	{1,3,4},{2}	31	23.5	28
5D_DS_Q19	{1,2,3,4,5}	5	12.6	40

Table 4.2: Comparison of query specific MSO Bound

4.3.2 MSO Bound

We will initially show the comparison of query specific MSO bounds for both the algorithms. In Table 4.2, π_r denotes the optimal partitioning for the given query template and t denotes the minimum tolerance required for aligning all the subcontours with partitioning π_r. The last two columns show the MSO bound for the two algorithms. For query template 19 with 5D `ESS`, the MSO bound comes down from 40 to 12.6, which is a significant reduction.

4.3.3 Empirical MSO

Let us see the empirical MSO values for `Opt-SB`. Here, we have used *local-partitioning*, where we find optimal partitioning for each contour as mentioned in Section 4.2. From the Table 4.3, we can infer that, even if we have comparable values of MSO for lower dimensions, we get improved MSO values when $D > 3$.

Table 4.4 shows the maximum value of tolerance and the number of partitions for each query template at any instance of `Opt-SB`. Hence, we confirm that the tolerance values are not very high in practice with sufficient partitions.

4.3.4 Average-case Performance

After comparing the two algorithms wrt worst-case performance, let us shift focus on the average-case performance. Since `Opt-SB` reduces the number of executions in each contour, we can see a consistent improvement in the ASO values for `Opt-SB` over `SpillBound` in Table 4.5.

Query Template	`Opt-SB`	`SpillBound`
2D_DS_Q96	5	5
3D_DS_Q15	7	6.6
3D_DS_Q96	9.6	9.8
3D_H_Q5	5.15	5
3D_H_Q7	6.1	6.4
4D_DS_Q7	6.7	14
4D_DS_Q91	7.8	7.8
4D_DS_Q26	9.7	13.2
5D_DS_Q19	7.8	12.9

Table 4.3: Empirical MSO using *local-partitioning*

Query Template	Tolerance(%)	#Partitions
2D_DS_Q96	60	2
3D_DS_Q15	67	2
3D_DS_Q96	61	3
4D_DS_Q7	69	2
4D_DS_Q91	45	3
4D_DS_Q26	43	2
5D_DS_Q19	41	2
3D_H_Q5	49	2
3D_H_Q7	39	2

Table 4.4: Empirical values of t and p

Query Template	`Opt-SB`	`SpillBound`
2D_DS_Q96	3.2	3.3
3D_DS_Q15	3.2	3.3
3D_DS_Q96	4.9	6.6
3D_H_Q5	2.9	3.3
3D_H_Q7	2.9	3.0
4D_DS_Q7	5.4	6.7
4D_DS_Q91	3.8	4.0
4D_DS_Q26	6.0	6.6
5D_DS_Q19	4.7	6.1

Table 4.5: Empirical ASO using *local-partitioning*

Chapter 5

Conclusion

In the initial work, we improve the MSO bound given by `PlanBouquet`, by utilizing the *cost acclivity assumption* on PIC, which mostly holds true in practice. We improved the MSO bound by removing its dependence on the plan density(i.e. ρ). In doing so, we substantially reduced the pre-processing time of finding the *bouquet* of plans.

In our second contribution, we investigate an intrusive technique called `SpillBound`, which explicitly learns selectivities by changing the execution module of the database engine. We proposed an optimized version of `SpillBound` called `Opt-SB`, wherein we execute bounded suboptimal plans at chosen selectivity locations, thereby maximizing the selectivity learning. We proved that `Opt-SB`'s MSO bound is in $O(Dp)$ which improves the $O(D^2)$ bound of `SpillBound`. Our experimental results, obtained on benchmark environments operating on PostgreSQL, demonstrate that `Opt-SB` performs well in practice with respect to both MSO and ASO metrics.

As a part of future work, we will try to get an MSO which has weak dependence on dimensionality D of `ESS`. Moreover, for higher dimensions, the total number of possible partitions become unreasonably large, which makes it infeasible to search for the optimal partitioning. In future, we will try to find a good heuristic technique which will prune the search space for finding the optimal partitioning.

Bibliography

[1] A. Dutt and J. Haritsa. Plan bouquets: Query processing without selectivity estimation. SIGMOD, 2014.

[2] S. Karthik, SpillBound, (Report in progress).

[3] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. VLDB, 2002.

[4] D. Harish, P. Darera, and J. Haritsa. Identifying robust plans through plan diagram reduction. *VLDB*, 2008.

[5] http://www.tpc.org/tpch/.

[6] http://www.postgresql.org/.

[7] http://dsl.serc.iisc.ernet.in/projects/EXPAND/html/doc_robust_4.html.

[8] http://mathworld.wolfram.com/BellNumber.html.