Design and Implementation Techniques for Plan Bouquet

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

Master of Engineering
IN
COMPUTER SCIENCE AND ENGINEERING

by
Sumit Neelam

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

JULY 2014
DEDICATED TO

My Family
for continuous support and encouragement
Acknowledgements

I am deeply grateful to Prof. Jayant 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.

Also, I am thankful to Anshuman Dutt for all those long discussions and suggestions. It had been a great experience to work with him.

My sincere thanks goes to my fellow lab mates 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 sisters 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

Given an SQL query, current database systems execute it using a plan that is estimated to be time optimal. In case of wrong estimation, execution process can be highly sub-optimal. Recently, an approach has been proposed to handle this issue, wherein the estimation process is completely avoided. Instead, a small set of plans is identified before execution. Then, at run time, the actual selectivities of the query are incrementally “discovered” through a sequence of partial executions of those plans. This approach assumes that database system has inbuilt facility to support monitoring and controlling mechanisms.

This work aims to design and implement techniques those are required to develop given approach. Specifically, these techniques comprise performing preprocessing before execution – identifying set of plans before execution; adding features in database engine to support controlling and monitoring mechanisms – cost (expected time) bounded partial execution, ability to exclusively execute specific part of a plan, and observing selectivity during execution; and developing external driver program that uses those controlling and monitoring mechanisms to execute query. In this work, PostgreSQL is used as database and experiments are run using TPC-H and TPC-DS benchmark queries.
Publications from the Thesis

Anshuman Dutt, Sumit Neelam and Jayant Haritsa
“QUEST: An Exploratory Approach for Robust Query Processing”
Proc. of 40th Intl. Conf. on Very Large Data Bases (VLDB), Hangzhou, China, September 2014.
## Contents

Acknowledgements i  
Abstract ii  
Publications from the Thesis iii  
Contents iv  
List of Figures vi  
List of Tables vii

### 1 Introduction
1.1 Motivation .............................................. 1  
1.2 Contributions ........................................... 2  
1.3 Organization ............................................ 2

### 2 Background
2.1 Query Processing Architecture .................................. 3  
2.2 Cost of Plans ............................................. 4  
2.3 Selectivity Estimation Problem .................................... 6  
2.4 Robust Query Processing ..................................... 6

### 3 Overview of Plan Bouquet
3.1 Single Dimension Example ................................... 7  
3.1.1 Bouquet Identification .................................... 7  
3.1.2 Bouquet Execution ...................................... 9  
3.2 Extension to Multiple Dimensions ............................... 9  
3.3 Robustness Metric ......................................... 10
# CONTENTS

4 Contributions

4.1 Compile Time mechanism ........................................ 12
4.2 Modifications to Database Engine ............................. 13
4.3 Execution Time mechanisms ................................... 13
4.4 QUEST Prototype System ...................................... 14

5 Efficient Contours Identification .............................. 15

5.1 2-Dimensional ESS with single contour ....................... 15
5.2 2-Dimensional ESS with multiple contours .................. 17
5.3 Experiments ..................................................... 19

6 Modifications to Database Engine ............................ 21

6.1 Cost Bounded Plan Execution ................................. 21
6.2 Cost Monitoring and Stopping Query Execution .............. 22
6.3 Sub-plan Execution ............................................. 23
6.4 Selectivity Monitoring ........................................ 25
6.5 Implementation Details ....................................... 26

7 Execution Time Mechanisms ................................... 27

7.1 Basic Bouquet Implementation ................................. 27
7.2 Optimized Bouquet Implementation .......................... 28
7.3 Implementation Details ....................................... 30

8 QUEST Prototype System ....................................... 31

8.1 Description ..................................................... 31
8.2 Features Details ............................................... 32
  8.2.1 Sub-optimality of Native Optimizer ...................... 32
  8.2.2 Bouquet Identification .................................... 33
  8.2.3 Bouquet Execution ........................................ 34
  8.2.4 MSO Guarantees ........................................... 35
  8.2.5 Optimized Bouquet ....................................... 36
8.3 Implementation Details ....................................... 36

9 Conclusions and Future Work ................................. 38

Bibliography ...................................................... 39
List of Figures

2.1 Example Query (EQ) ................................................. 3
2.2 Query Plan for EQ .................................................. 4
2.3 Calculating $n$ values .............................................. 5

3.1 POSP Plans ............................................................. 8
3.2 POSP performance ................................................... 8
3.3 2-D Selectivity Space ............................................... 10

5.1 1-D Contours Identification ........................................ 15
5.2 Single Contour Identification ...................................... 16
5.3 Multiple Contours Identification ................................... 17
5.4 Contours and Explored Points Plots ............................... 20

6.1 Cost Bounded Execution ............................................ 23
6.2 Sub-plan Execution .................................................. 25
6.3 Selectivity Monitoring .............................................. 26
6.4 Sub-plan Execution Flow ........................................... 26

7.1 Basic Bouquet Execution Process ................................. 28
7.2 Basic Bouquet Example ............................................. 28
7.3 Optimized Bouquet Execution Process ............................ 29
7.4 Optimized Bouquet Example ...................................... 30

8.1 QUEST Architecture ................................................ 32
8.2 Sub-optimality of Native Optimizer ............................... 33
8.3 Bouquet Identification Interface .................................. 34
8.4 Bouquet Execution Interface ...................................... 35
8.5 Optimized Bouquet with Selectivity Monitoring ................ 36
List of Tables

5.1 Contours Identification Experiments Results .................................. 20
6.1 Counters for Cost Estimation ......................................................... 22
Chapter 1

Introduction

Data processing in current database systems is achieved using SQL queries. Since the SQL queries are declarative, query optimizer module considers different ways or Plans to execute the query and evaluates each plan by estimating its expected cost. Cost of different plans give relative measure of expected running time of those plans. Estimated cost of the plan is function of Selectivity which is fraction of the number of rows of each relation relevant in producing the final result. After evaluation, the minimum cost plan is chosen for execution and passed to executor module.

1.1 Motivation

Database optimizer estimates number of selectivities during plan selection for SQL queries. In practice, these estimates are different with respect to actual values encountered during query execution and that may result in sub-optimal execution performance. In [3], a different approach is proposed for handling such error-prone selectivity. It completely skips error-prone selectivity estimation during plan selection, rather query is executed through a calibrated sequence of cost bounded plan executions. The set of plans used in the sequence is termed as “Plan Bouquet”. The main feature of plan bouquet approach is that it provides theoretical performance bounds for query execution process. Moreover, it shows those bounds can be improved by utilizing run time information using selectivity monitoring and sub-plan execution techniques.

Through design and implementation techniques, plan bouquet approach can be realized. With actual running plan bouquet approach, performance bounds can be validated empirically and effectiveness of selectivity monitoring and sub-plan execution techniques can be tested for improving bounds.
1.2 Contributions

In this work, we discuss the design and implementation of techniques required to have plan bouquet in existence. These techniques include preprocessing – to efficiently identify plans of plan bouquet; developing features inside database engine such as – cost bounded partial execution, sub-plan execution, selectivity monitoring.

In cost bounded partial execution, query is executed up to provided cost threshold. Query execution operation spends time in CPU and I/O operations. So during cost limited execution, CPU and I/O operations are continuously counted and based on number of operations, cost is progressively calculated. When run time calculated cost crosses cost threshold, query execution is stopped and all results, generated by partial execution, are thrown. In sub-plan execution, execution is limited to a specific part of plan tree, i.e. a part of plan tree is executed without executing other remaining part. In selectivity monitoring, selectivity of node is continuously examined by counting tuples produced by node.

Along with these database engine features, plan bouquet algorithm is implemented to execute query by utilizing extra features added to database engine and a visual interface is developed that provides interactivity during query execution.

1.3 Organization

The remainder of this thesis is organized as follows: Chapter 2 provides the background details that includes traditional query processing concepts with selectivity estimation problem and brief description of robust query processing. Overview of plan bouquet technique is given in Chapter 3. Chapter 4 briefly discusses contributions in terms of required mechanisms for plan bouquet. Detailed design methodology with implementation details of required mechanisms are given in Chapters 5, 6, and 7. Practical demonstration of this work is presented in Chapter 8. Finally, in Chapter 9, conclusion of this work is summarized and outline of future work is provided.
Chapter 2

Background

In this chapter, we present necessary details about query processing in current database systems, selectivity estimation problem and robust query processing.

2.1 Query Processing Architecture

In query processing operation, initially, query is checked for syntax and semantic errors then it is passed to optimizer module. Optimizer considers different Plans to execute the query and evaluates those plans by estimating their expected cost which is estimation of expected running time of query with that plan. Expected plan cost is calculated by estimating selectivities involved in query. Among those plans, optimizer selects minimum cost plan for execution. This phase is called plan selection or optimization phase. Minimum cost plan is then passed to executor where actual execution takes place.

For example, consider EQ given in Figure 2.1, which contains three relations lineitem, orders, part. Here, optimizer needs to estimate selectivities of one selection predicate (\( p_{\text{retailprice}} \)) and two join predicates (\( \text{part} \bowtie \text{lineitem} \) and \( \text{lineitem} \bowtie \text{orders} \)). After optimization phase, optimizer selected plan is shown in Figure 2.2.

**Query Plan and its Execution:** A query plan is tree structured implementation that states detailed way of query execution. Each node of tree is called plan node that encapsulates a single operation that is required to execute the query. The leaves of the tree generate tuples...
by scanning relations of query. Non-leaf nodes get tuples from their child nodes, and after processing those child nodes tuples, themselves generate tuples.

In PostgreSQL database [7], root node requests tuple from its child nodes. Child nodes further request tuples from their child nodes and this process continues down to leaf nodes. That is, request of tuples flows down to leaf nodes starting from root node. Tuples are passed in plan tree in reverse order of request flow, i.e. from leaf nodes to root node.

In plan shown in Figure 2.2, leaf nodes of plan are scan operators and get tuples by scanning lineitem, orders and part relations. Non-leaf nodes HJ and Hash get tuples from their child nodes and after processing child nodes tuples, pass tuples to their parent nodes. Tuples generated by root node NL are shown as result of query.

Pipeline, Blocking and Non-Blocking Nodes: In a query plan, the concurrent execution of a contiguous sequence of plan nodes is referred as “Pipeline”.

In the context of pipeline, nodes in plan tree are divided into two groups, Blocking and Non-Blocking nodes. A node is called blocking node if it does not produce any output tuple until it has fully executed sub-tree on which it is rooted. In other words, blocking nodes break pipeline in plan execution. Non-blocking nodes concurrently execute in sequence without breaking pipeline. Hash, Sort nodes are examples of blocking nodes.

Query plan shown in Figure 2.2 contains two pipelines and shown in dotted shapes. Hash node divides plan into two pipelines. Hence, execution of sub-plan tree rooted on Hash (Pipeline 1) is completed first before passing control to other pipeline.

2.2 Cost of Plans

Cost of a plan is assessment of work done by plan. In calculation of cost, number of operations performed by plan is determined and multiplied by single operation costs.
Cost of plan described here is according to cost model of PostgreSQL database [7]. PostgreSQL maintains a vector $c=[c_s, c_r, c_t, c_i, c_o]^{\top}$ of values known as unit costs or per operation costs. Where

- $c_s$: Sequential Page Cost
- $c_r$: Random Page Cost
- $c_t$: Per Tuples Cost
- $c_i$: Per Index Tuple Cost
- $c_o$: Per Tuple Operation Cost

During cost calculation in PostgreSQL, two vectors are multiplied using dot product at each plan node.

\[
\text{Cost of plan} = \sum_{p \in S} n_p^{\top} c_p
\]  

where $S$ = set of plan nodes

For $p^{th}$ plan node, $n_p = [n_s, n_r, n_t, n_i, n_o]^{\top}$ and

- $n_s$: Number of Sequential Page Accesses
- $n_r$: Number of Random Page Accesses
- $n_t$: Number of Tuple Accesses
- $n_i$: Number of Index Tuple Accesses
- $n_o$: Number of Tuple Operations

Hence, vector $n$ holds values for number of operations and vector $c$ holds costs of each single operation.

During plan selection, optimizer foresees values of vector $n$ by estimating selectivity. Optimizer predicts selectivity that represents expected tuple accesses and other values of $n$ such as page accesses, tuple operations are determined as function of tuple accesses as shown in Figure 2.3.

![Figure 2.3: Calculating n values](image-url)
2.3 Selectivity Estimation Problem

As we saw, database optimizers estimate number of selectivities while identifying the ideal execution plan for SQL queries. In practice, these estimates are often significantly in error with respect to the actual values subsequently encountered during query execution. Such errors arise due to a variety of reasons including outdated statistics, attribute-value independence assumptions, coarse summaries, complex user-defined predicates. Such erroneous selectivities are called Error-prone selectivities and space formed by such selectivities is called Error-prone Selectivity Space (ESS).

2.4 Robust Query Processing

To solve selectivity estimation problem, many techniques are evolved such as improving the quality of statistical meta data, feedback based adjustment [2] etc. A conceptually different approach is Robust Query Processing where aim is to provide performance that is less sensitive to such selectivity estimation errors. In a nutshell, to “aim for resistance, rather than cure”, by providing technique that can cope with selectivity estimation problem. Plan bouquet approach also provides robustness through careful execution process and its robustness metric is given in next chapter.
Chapter 3

Overview of Plan Bouquet

Plan Bouquet [3] is a new approach, wherein the estimation process is completely skipped for error-prone selectivities. Instead, these selectivities are systematically discovered at run-time through a calibrated sequence of cost bounded plan executions.

In whole discussion of bouquet, Plan Cost Monotonicity (PCM) assumption is made which states that cost of POSP plans increase monotonically with increasing selectivity values.

Plan Cost Monotonicity: For 2-D selectivity space, consider two points \( q_1(x_1, y_1) \) and \( q_2(x_2, y_2) \) and a plan \( P \). Suppose \( C(P, q_1) \) and \( C(P, q_2) \) show costs of executing \( q_1(x_1, y_1) \) and \( q_2(x_2, y_2) \) with plan \( P \) receptively. Then cost of executing \( q_2 \) with plan \( P \) will be greater then cost of executing \( q_1 \) with plan \( P \) if

\[
x_2 > x_1 \text{ and } y_2 > y_1 \text{ or } \\
x_2 = x_1 \text{ and } y_2 > y_1 \text{ or } \\
x_2 > x_1 \text{ and } y_2 = y_1.
\]

3.1 Single Dimension Example

Here, plan bouquet is explained for one error-prone selectivity using query shown in Figure 2.1. Given query contains three selectivities (one predicate and two join selectivities). Out of these three selectivity, \( p_{\text{retailprice}} \) selectivity is error-prone.

3.1.1 Bouquet Identification

First, through repeated invocations of the optimizer, we identify the “parametric optimal set of plans” (POSP) that covers the entire selectivity range of the predicate, i.e. by varying
p\_retailprice from its minimum to maximum value POSP is discovered. A sample outcome of this process is shown in Figure 3.1, wherein the POSP set is comprised of plans P1 through P5. Further, each plan is annotated with the selectivity range over which it is optimal – for instance, plan P2 is optimal in the 0.3% to 1.0% selectivity range.

![Figure 3.1: POSP Plans](image)

The costs of these five plans, P1 through P5, over the selectivity range are enumerated in Figure 3.2, using abstract plan costing. Through abstract plan costing, cost of a plan can be determined anywhere in selectivity space. From these plots, we derive the “POSP Infimum Curve” (PIC), defined as the trajectory of the minimum cost from among the POSP plans – this curve represents the ideal performance.

![Figure 3.2: POSP performance](image)

The next action, which is a distinctive feature of the bouquet approach, is to discretize the
PIC by projecting a graded progression of isocost (IC) steps onto the curve. For example, in Figure 3.2, 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 IC with the PIC (indicated by ■) provides an associated selectivity, along with the identity of the best POSP plan for this selectivity. For example, in Figure 3.2, the intersection of IC5 with the PIC corresponds to a selectivity of 0.65% with associated POSP plan P2. The subset of POSP plans associated with the intersections form the “plan bouquet” for the given query – in Figure 3.2, the bouquet consists of P1, P2, P3, P5.

3.1.2 Bouquet Execution

After identifying plan bouquet, partial execution of plans are done in increasing order of cost limit. So in previous example, execution sequence of plans comes out as P1, P2, P3, P5.

It should also be noted that selectivity corresponding to IC4 (9.6E4) is 0.2% and IC4 intersects with P1. So using PCM property, query having selectivity less then 0.2% can be finished inside cost 9.6E4 with plan P1. Similar argument can be made for other plans.

The bouquet execution is now presented through a example query instance where the selectivity of \( p_{retailprice} \) is 2.5%. We begin by partially executing plan P1, corresponding to the cheapest isocost step IC1, until the execution overheads reach IC1 (1.2E4 \( \mid 0.015\% \)). Then, we extend our cost horizon to IC2, and continue executing P1 until the overheads reach IC2 (2.4E4 \( \mid 0.03\% \)), and so on until the overheads reach IC4 (9.6E4 \( \mid 0.2\% \)). At this juncture, there is a change of plan to P2 as we look ahead to IC5 (1.9E5 \( \mid 0.65\% \)), and during this switching all the intermediate results (if any) produced thus far by plan P1 are jettisoned. The new plan P2 is executed till the associated overhead limit (1.9E5) is reached. The cost horizon is now extended to IC6 (3.8E5 \( \mid 6.5\% \)), in the process jettisoning plan P2’s intermediate results and executing plan P3 instead. In this case, the execution will complete before the cost limit is reached since the actual location, 2.5%, is less than the selectivity limit of IC6.

3.2 Extension to Multiple Dimensions

In multi-dimensional selectivity environment, the IC steps and the PIC curve become surfaces, and their intersections represent selectivity surfaces on which multiple bouquet plans may be present. For example, in the 2-D case, the IC steps are horizontal planes cutting through a hollow 3-D 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 3.3(a).

In single dimension, we jump from one contour to next contour after executing single plan.
(because only one plan lies on each contour in 1-D case). But in multiple dimensions, we can jump from one contour to next contour only after executing each plan on contour or by using some optimizations.

In Figure 3.3(b), shaded area under plan curve of $P_{10}$ of contour $IC_3$ is shown. As an implication of PCM property, cost of every point inside shaded area will be less than cost of $IC_3$, which indicates that plan $P_{10}$ can finish any query lying in shaded region within cost limit of $IC_3$. This property is called “Third Quadrant Coverage” for plans.

### 3.3 Robustness Metric

The notion of robustness is given by maximum sub-optimality (MSO) for plan bouquet. Given a query $Q$ and error selectivity space $ESS$, 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 $\text{cost}(P_j, q_i)$ represent the execution cost incurred at actual location $q_i$ by plan $P_j$. Then, robustness is defined by the normalized metric:

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

which ranges over $[1, \infty)$. Plan bouquet gives robustness guarantee in terms of MSO as:

$$MSO \leq 4\rho$$
If there exist K total contours and \( n_i \) plans lie on \( i^{th} \) contour then,

\[
\rho = \max_{i=1 \text{ to } K} n_i
\]

It is clear from 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 is possible through anorexic
reduction [1] which decreases number of plans in POSP without substantively affecting the
query processing quality of any individual query in the selectivity space.
Chapter 4

Contributions

We now turn our attention to contributions of this work which describes required mechanisms for the plan bouquet. Here, these mechanisms are described with their brief introductions. In later chapters, these mechanisms are discussed in detail with their design and implementation. These mechanisms can be divided into three categories:

- Compile Time mechanism
- Modifications to Database Engine
- Execution Time mechanisms

Apart from these mechanisms, a graphical interface is also implemented that integrates these mechanisms.

4.1 Compile Time mechanism

This mechanism is done prior to execution so can be considered as preprocessing for actual execution. It includes finding contours spread across ESS and plans lying on those contours.

Efficient Isocost Contours Identification: An isocost contour with cost $C$ is all locations in ESS having cost equal to $C$. Here we identify a set of contours assigned cost in geometric progression with common ratio 2. Initially, cost of ESS locations are unknown. So for finding cost of a location in ESS, database engine is queried. In the process of finding cost, database engine performs optimization and find cost along with plan optimal at that location.

Currently, brute-force approach is used to identify contours, wherein cost of each ESS location is identified and then contours’ locations are searched in entire ESS, i.e. each point in ESS
is optimized. Time required to do this optimization and to find cost is noteworthy. Therefore, this overall process is time consuming.

In this work, we give an algorithm that identifies contours’ locations without exploring each point of ESS. Formally, it can be described as follows:

Given ESS and contours’ cost set C, Contour Identification finds all locations in ESS having cost equal to one of the elements of C.

4.2 Modifications to Database Engine

Plan bouquet expects some inbuilt controlling and monitoring mechanisms inside database engine for running queries. In this work, these mechanisms are implemented inside database engine. Here, these are described in brief.

Cost Bounded Plan Execution: In this mechanism, query execution is done up to a given cost bound, i.e. a cost threshold is also given as input along with query and that serves as bound for query execution. Formally, it can be defined as follows:

Given a query plan P and cost bound C, Cost Bounded Plan Execution monitors query execution process through tracking of cost and stopping execution when tracked cost passes bounded cost C.

Sub-plan Execution: Sub-plan Execution allows to execute a subpart of plan tree without executing other parts. Formally, it can be described as:

Given a query plan P and a plan node N of P, Sub-plan Execution exclusively executes sub-plan rooted at node N.

Selectivity Monitoring: Selectivity Monitoring provides learning from partial plan executions. Formally it can be described as:

Given a query plan P and plan node N, Selectivity Monitoring continuously counts tuples produced by given node N and computes selectivity.

4.3 Execution Time mechanisms

Execution time mechanisms include implementing a external driver program that uses modified database engine for executing queries with plans found at compile time. The driver program implements basic bouquet and optimized bouquet algorithms.

Implementation of Plan Bouquet Algorithms: In this work, two variants of plan bouquet algorithms are implemented – Basic bouquet and Optimized bouquet. Basic bouquet uses only cost bounded plan execution feature but optimized bouquet improves performance by utilizing selectivity monitoring to get learning from previous partial executions.
4.4 QUEST Prototype System

For visually showing bouquet execution process a prototype implementation QUEST (QUery Execution without Selectivity esTimation) is developed. QUEST is a graphical interface that incorporates plan bouquet algorithms and shows execution process. It also demonstrates selectivity estimation problem in current database and bouquet identification process visually.

In following chapters, these mechanisms are described in detail with their design issues.
Chapter 5

Efficient Contours Identification

Initially, contours identification method is explained for 2-dimensional ESS having single contour. Later, this idea can be extended for multi dimensions and multi contours. Here, this idea assumes that Plan Cost Monotonicity (PCM) property is followed throughout ESS, i.e. cost increases monotonically with increase in selectivity values.

In 1-D case, single plan exists on each contour and ESS is limited to single line as given in Section 3.1 of Chapter 3. 1-D contours identification process is shown in Figure 5.1 where contours are shown as points. Figure shows cost of minimum and maximum selectivity points as \( C_{\text{min}} \) and \( C_{\text{max}} \) respectively and cost on this line increases monotonically from \( C_{\text{min}} \) to \( C_{\text{max}} \). First maximum cost point (contour) is searched using binary search kind of algorithm between \( C_{\text{min}} \) and \( C_{\text{max}} \). This point is shown as \( C_1 \). Further, next higher cost contour point is found between \( C_{\text{min}} \) and \( C_1 \) which is shown as point \( P_2 \). Same process is continued for finding other points.

![Figure 5.1: 1-D Contours Identification](image)

5.1 2-Dimensional ESS with single contour

Figure 5.2(a) shows a contour of cost C. Contour identification process starts with searching a point of cost C on principal diagonal of ESS. In presence of PCM property, we can use binary search kind of algorithm to find such point because cost will increase along diagonal from
bottom-left corner to top-right corner. Such point on principal diagonal is shown by P1(x₁,y₁) in Figure 5.2(b). If dimensional-axes are drawn corresponding to this point as origin then ESS will be divided into 4 subspaces as shown in same Figure. Using PCM property, we can argue that contour will not lie in 1st and 3rd subspaces. It can be illustrated using two cases shown in Figures 5.2(c) and 7(d).

![Diagram showing single contour identification](image)

**Figure 5.2: Single Contour Identification**

In Figure 5.2(c), contour is shown which lies in 1st subspace also. Here, point P2(x₂, y₂) is also shown which lies below contour. All points lying on contour have cost C and points lying below contour have cost less than C. Hence,

\[ \text{Cost}(P2) < C \]

Since P2 lies in 1st quadrant with respect to P1 therefore, x₂ > x₁ and y₂ > y₁. So according to PCM, cost at point P2 should be greater than cost at point P1 which is C. This violates PCM property. Similar argument can be given for Figure 5.2(d). As a result of this argument,
we need to search remaining part of contour in 2\textsuperscript{nd} and 4\textsuperscript{th} subspaces only.

Further exploration process is shown in Figures 5.2(e) and 5.2(f) where subspaces are formed including dimensional-axes points considering P1 as origin, i.e. boundary points of subspaces are also included in exploration. These subspaces are further divided into subspaces using principal diagonal. In the course of finding contours, resolution is kept fixed, i.e. number of points are fixed in ESS through which contours can pass.

So this technique is summarised as – first intersection point between contour and ESS principal diagonal is found that divides ESS into four subspaces then, recursively contour is searched in 2 subspaces (2\textsuperscript{nd} and 4\textsuperscript{th}) and remaining 2 subspaces are rejected.

![Figure 5.3: Multiple Contours Identification](image)

**5.2 2-Dimensional ESS with multiple contours**

Now, this idea can be extended for 2-D ESS and multiple contours those are in geometric progression. Figure 5.3 shows ESS with multiple contours. Here, identification process starts with finding costliest contour and then moving to cheaper contours. In the space, initially a point is found on principal diagonal having cost equals to costliest contour’s cost of that space. 1\textsuperscript{st} subspace with respect to that point can be completely rejected. Unlike 2-D ESS single contour instance, here 3\textsuperscript{rd} subspace can contain cheaper contours. Hence, out of four subspaces one subspace is rejected and remaining three are explored recursively including dimensional-axes points.

In general, for d-dimensional ESS, a point divides ESS into 2\textsuperscript{d} subspaces out of which 2\textsuperscript{d} - 1 need to be explored further.
Algorithm 1 Contours Identification Algorithm

FindContours(ESS, bottom, top, maxCost, d)

\( r = 2 \) /* geometric progression common ratio */
/* base case for recursion */
if \( top = bottom \) then
/* subspace will contain single point */
top_point_cost = ESSPointCost(ESS, top)
if \( maxCost = top\_point\_cost \) then
    AddContourLocation(ContoursLocation, top\_point, maxCost)
end if
return
end if

bottom_point_cost = ESSPointCost(ESS, bottom)
top_point_cost = ESSPointCost(ESS, top)

/* maximum contour cost value is found that lies in current subspace */
while \( maxCost > top\_point\_cost \) do
/* cost is divided by 2 because cost ratio is 2 between contours */
    maxCost = maxCost/r
end while
if \( maxCost < bottom\_point\_cost \) then
/* reject region because no contour lies in this subspace */
    return
end if

p = FindPoint(ESS, bottom, top, maxCost, d)
AddContourLocation(ContoursLocation, p, maxCost)

/* calculating bottom-left and top-right corners for each subspace */
for \( i = 1 \) to \( 2^d \) do
    using points bottom, top, and p, calculate \( top_i \) and \( bottom_i \) for \( i^{th} \) subspace
    if \( top_i = top \) then
        /* only single subspace satisfies this condition */
        /* do nothing */
    else
        /* remaining \( 2^d - 1 \) subspaces */
        FindContours(ESS, bottom\_i, top\_i, maxCost, d)
    end if
end for
Pseudo code for identifying contours is given in Algorithm 1. Algorithm takes five arguments – \( ESS \) is space where contours are supposed to be found; \( bottom \) and \( top \) are coordinates of bottom-left and top-right corners of \( ESS \); \( maxCost \) is largest cost among all contours’ cost; \( d \) is number of dimensions of \( ESS \).

Algorithm contains statements of comparing two coordinate points, e.g. \( p_1 = p_2 \), which means equality is checked between points \( p_1 \) and \( p_2 \) for each dimension.

**FindPoint**\((ESS, P_1, P_2, C, dimension)\): This function returns point having cost \( C \) on line joining points \( P_1 \) and \( P_2 \) in \( ESS \).

**AddContourLocation**\((ContoursLocation, P, C)\): This function stores cost \( C \) for given coordinate \( P \) in global data structure \( ContoursLocation \).

**ESSPointCost**\((ESS, P)\): This function returns cost of \( ESS \) point \( P \). Actually, this function queries database optimizer to find cost of given point.

**Algorithm Description:** Initially base case for recursion is checked. Then using top-right point cost, contour’s cost is found that is maximum inside that subspace, i.e. if current contour does not lie in a subspace then cheaper contours are searched in that subspace. Further, point on principal diagonal is found having contour’s cost. Using this point, space is divided into subspaces and subspaces are searched recursively. In later part of algorithm, \( top_i = top \) is used for rejecting single region. After finishing algorithm contours locations can be found in \( ContoursLocation \) data structure.

### 5.3 Experiments

Currently, this algorithm is not implemented for finding contours but externally evaluated. For evaluating this algorithm, experiments are done on different cost functions which are smooth and monotonically increasing. Here, results are shown for three cost functions:

\[
\begin{align*}
  f_1 &: x^2 + y^2 \\
  f_2 &: x \cdot y \\
  f_3 &: x^2 + y^2 + z^2
\end{align*}
\]

For these cost functions five contours are identified with contours’ cost in geometric progression of common ratio 2. Resolution is kept fixed to 100 for each dimension.

In Figure 5.4, plots for contours and explored points are given for 2-D cost functions \( f_1 \) and \( f_2 \). Figure 5.4(a) shows locations where contours lie and Figure 5.4(b) shows locations explored in \( ESS \) for finding contours for \( f_1 \). Similarly, both plots are shown for \( f_2 \) in Figures 5.4(c) and 5.4(d). Results for three functions are shown in Table 1.

Table 1 gives dimension, total points in \( ESS \), number of points explored for finding contours.
and contours points for each function. A brute-force algorithm will find contours by exploring all points in ESS (Total Points). This algorithm finds contours by exploring points given in Explored Points column.
Chapter 6

Modifications to Database Engine

In this chapter, design issues of extra features required for bouquet execution inside database engine are discussed in detail.

6.1 Cost Bounded Plan Execution

In cost bounded plan execution, cost of query is continuously monitored at execution time and query execution is stopped when monitored cost crosses given threshold value.

Cost monitoring operation uses Equation 1 applied at plan selection time which involves vector $c$ (single operations cost values) and vector $n$ (total number of operations). For cost monitoring, same vector $c$ is used as optimization time. But after optimization phase, these values are relinquished and no more available at run time. So, this vector is transferred from optimization time to run time.

Transferring vector $c$ cost values: In plan tree, these costs can be different for some nodes so these values maintained per node basis and at plan selection time, costs are explicitly stored in each plan node. When execution plan tree is built, same explicit values are copied to execution time plan tree. This is done by modifying data structures of optimization plan tree and run time plan tree.

Unlike vector $c$, vector $n$ is calculated by counting number of operations because during optimization phase, $n$ is calculated through estimated selectivities which can be different from actual selectivities.

Counting number of operations: Vector $n$ is calculated using some implicit and explicit counters. In Table 2, counters are shown with implicit availability in PostgreSQL 8.3. Counters, which are not implicitly available, are incremented in different files of PostgreSQL where corresponding operations are performed.
<table>
<thead>
<tr>
<th>Name</th>
<th>Default Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq. Page Access Counter</td>
<td>no</td>
</tr>
<tr>
<td>Random Page Access Counter</td>
<td>no</td>
</tr>
<tr>
<td>Tuple Counter</td>
<td>yes</td>
</tr>
<tr>
<td>Index Tuple Counter</td>
<td>yes</td>
</tr>
<tr>
<td>Tuple Operation Counter</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 6.1: Counters for Cost Estimation

Counters are maintained at global level using fact that PostgreSQL executes on single processor, hence race condition is not possible on these counters.

In multi processors database system, two approaches can be used – first, by applying locking on counter variables and second, by using multiple counter variables. In the first approach, lock is applied before accessing counter variable to cope with synchronize problem and lock is released after using it. But with this approach, database performance will be affected, since processes will have to wait for releasing of lock.

In the second approach, counter variables are maintained for each plan node. So, two or more plan nodes can be executed in parallel without locking problem of counter variable.

6.2 Cost Monitoring and Stopping Query Execution

Cost monitoring is triggered after a tuple is processed by a plan node. Cost of processing a tuple is determined by fetching counter value of operations and then multiplying by unit costs. Pseudo code for cost bounded execution is given separately for non-blocking and blocking node in Algorithms 2 and 3 respectively.

In given two algorithms, `estimate_cost()` function calculates cost of producing the tuple from plan node using Equation 1. `cost_bound` and `monitored_cost` hold threshold cost (limit cost) value and run time monitored cost of query at global level.

Both algorithms provide an abstract view of plan nodes execution; e.g. a join node has two child nodes and gets one tuple from each child. But in pseudo code, this is encapsulated in single call of `ProcessNode()`.

Example of Cost Bounded Execution: A query plan, shown in Figure 6.1, is executed with cost bounded execution feature. Suppose, `monitored_cost` increases from `cost_bound` when tuple is processed at node3 (NL). So inside `ProcessNode()` for node3, `node3 \rightarrow execution_finished` is assigned `true`. When control is passed to `ProcessNode()` of node2 (HJ), it observes that `(node2 \rightarrow child) \rightarrow execution.finished`, i.e. `node3 \rightarrow execution.finished` is `true`. It pretends to node2 that execution at node3 is finished and node3 does not have tuples to deliver. Hence,
Algorithm 2 Non-Blocking Node Execution

ProcessNode(node)

/*a single tuple from child node is fetched*/
t = ProcessNode(node → child)
if node → child → execution_finished = true then
    node → execution_finished = true
    return NULL
end if

tuple = process_tuple(t)

/*cost of processing tuple is calculated and monitored cost is checked*/
monitored_cost = monitored_cost + estimate_cost(node, tuple)
if monitored_cost > cost_bound then
    node → execution_finished = true
    return NULL
end if
return tuple

Figure 6.1: Cost Bounded Execution

node2 also assigns true to node2 → execution_finished. Same process is done recursively on other nodes and whole query execution is stopped.

6.3 Sub-plan Execution

Current database systems start query plan execution with their root node. In sub-plan execution feature, query plan execution starts at particular node, which is specified in input.

This technique is useful when we want to determine selectivity of a particular node. In the normal execution, entire plan tree needs to be executed for doing this operation. But with this technique, given operation can be done by executing only required number of plan nodes.

Sub-plan execution pseudo code is given in Algorithm 4. In the algorithm, SearchPlanNode()
Algorithm 3 Blocking Node Execution

ProcessNode(node)

/*execute till child node finishes execution or monitored_cost exceeds cost_bound*/
while node → child → execution_finished = false do
    t = ProcessNode(node → child)
    store t in tuple_store
end while

/*extra condition is checked*/
if monitored_cost > cost_bound then
    node → execution_finished = true
    return NULL
end if

tuple = process_tuple(tuple_store)
monitored_cost = monitored_cost + estimate_cost(node, tuple)
if monitored_cost > cost_bound then
    node → execution_finished = true
    return NULL
end if
return tuple

Algorithm 4 Sub-plan Execution

Sub-planExecution (root_ref, node_id)

node_ref = SearchPlanNode(root_ref, node_id)

/* NULL shown absence of node_id in plan tree*/
if node_ref ≠ NULL then
    new_root_node = node_ref
else
    error Invalid Plan node
end if
return new_root_node

is the standard node search algorithm for trees.

Example of Sub-plan Execution: In Figure 6.2, a query plan is shown along with ID and REF of each node. Nodes in plan tree can be uniquely identified using their IDs and REF is used to get information required to execute a node. If sub-plan shown inside dotted circle is to
be executed without execution of full plan; given algorithm is called with reference of root node (REF1) and ID of sub-plan root node (ID3). Given algorithm will return reference of sub-plan root node (REF3). As a result of giving reference of NL (REF3) to executor, sub-plan rooted on ID3 will be executed.

6.4 Selectivity Monitoring

In presence of predicate with a plan node, some tuples may not satisfy that predicate. Under that case, selectivity of that node will be fraction of tuples that will satisfy the predicate and will be produced by plan node. Selectivity of a join and single child or leaf node is calculated using following equations:

For join node,

\[
selectivity = \frac{\text{tuples produced by node}}{\text{tuples produced by child}_1 \times \text{tuples produced by child}_2}
\]

For single child node or leaf node,

\[
selectivity = \frac{\text{tuples produced by node}}{\text{tuples produced by child node or scanning from disk}}
\]

Selectivity monitoring keeps observing selectivity during execution of query and helps in optimizing plan bouquet.

During selectivity monitoring of node N, if all descendants of N are free from selectivity error, then optimizer estimated values of tuples are used for descendants and actual value is used for N. But if a descendant node is error prone node, then selectivity of that node is monitored separately and then that value is used in selectivity monitoring for node N.
In Figure 6.3, a part of plan tree is shown along with tuples produced by each node. Suppose, nodes $A$ and $B$ are free from selectivity errors, then

$$\text{selectivity of } MJ = \frac{500}{100 \times 300} = 0.0166$$

6.5 Implementation Details

All above described features are implemented through modification of PostgreSQL 8.3. For cost bounded execution, unit cost values are passed from optimizer phase to executor phase by modifying $path$ and $plan$ structures. Page access counters are incremented in $fd.c$ which contains file operation functions. For counting tuples, index tuples, tuple operations different node executor files are modified such as $execScan.c$, $nodeIndexScan.c$, $nodeNestedLoop.c$ etc. Execution time costing is done using new cost functions written in $executionCost.c$. Execution cost is stored on each plan node in two variables $startupCost$ and $totalCost$ (by modifying $PlanState$ structure).

In sub-plan execution, sub-plan module is put between optimizer module and executor module. So instead of passing plan tree reference directly to executor, it is passed through sub-plan execution module. This module can pass reference of any node in tree based on sub-plan root id given to it. These modifications are done in $execMain.c$ file.

Figure 6.4: Sub-plan Execution Flow
Chapter 7

Execution Time Mechanisms

In this chapter, bouquet algorithms’ design methodology is discussed which is implemented inside driver program that executes query on modified database engine.

7.1 Basic Bouquet Implementation

The basic bouquet approach performs query execution process by executing each plan on each contour in cost bounded manner until one of them completes the query. This process starts with cheapest contour and moves to higher cost contours.

Initially, an execution sequence of plans is decided for each contour. Then execution is begun by executing cheapest contour’s plans in decided sequence with cost bounded plan execution feature. For each plan, the driver program sends request to database for executing query with given plan and given cost limit. In reply, database sends query completion information back to driver program. If query is not completed, next plan in sequence of current contour is executed. If all plans on current contour are finished then control is passed to next contour. In this execution, cost bounded plan execution is only extra required feature from database engine. This execution process is shown in Figure 7.1.

Example: In Figure 7.2, 2-D ESS is shown along with 5 contours. A query is given for execution whose selectivity is shown by $q_a$ and lies under plan $P_3^3$ of contour 3. Suppose we decide following plans execution order for contours:

Contour $IC_1$: $P_1^1$, $P_2^1$
Contour $IC_2$: $P_1^2$, $P_2^2$, $P_3^2$, $P_4^2$, $P_5^2$
Contour $IC_3$: $P_1^3$, $P_2^3$, $P_3^3$, $P_4^3$
Contour $IC_4$: $P_1^4$, $P_2^4$
Contour $IC_5$: $P_1^5$
Execution is begun by executing $P_1^1$ with cost limit of $IC_1$. Using third quadrant coverage property given in Section 3.2 of Chapter 3, we can argue that plans of contours $IC_1$ and $IC_2$ will not be able to finish query in assigned cost budget. Using same argument, plan $P_1^3$ and $P_2^3$ will not finish query execution and in each of these executions, database will reply with uncompleted query execution status. Finally, query execution is finished by plan $P_3^3$ within cost limit $IC_3$.

7.2 Optimized Bouquet Implementation

Bouquet execution is optimized by selectivity monitoring during execution process. Selectivity monitoring entitles to learn actual selectivity from each partial execution. In the course of execution, selectivity is monitored such that it serves as lower bound for actual selectivity. The
strategy used is to ensure that at all times, the actual selectivity location is in the first quadrant with respect to the current selectivity location as origin. Hence, some plans can be skipped from execution those do not lie in first quadrant with respect to current selectivity.

Initially, monitored selectivity $q_{run}$ is initialized with origin of ESS and location of $q_{run}$ is incrementally updated after each (partial) plan execution.

Unlike the basic bouquet, in optimized bouquet execution sequence of plans is not fixed. Here, one of the axis plans of $q_{run}$ is chosen for execution. Axis plans are the set of bouquet plans present at the intersection of the isocost contour with the dimensional-axes corresponding to $q_{run}$ as origin. For finding axis plans, we need to have information of coordinate of each contour along with plans in ESS. So before starting of execution, contours location information is stored in a data structure. During execution, plans are executed with partial execution mechanism similar as basic bouquet. But here database replies with partial plan execution information that helps in learning selectivity and updating $q_{run}$. This execution process is shown in Figure 7.3.

![Figure 7.3: Optimized Bouquet Execution Process](image)

During execution of optimized bouquet, selectivity monitoring feature is required from database along with cost bounded plan execution feature. Moreover, amount of learning can be further increased by ensuring that the cost is used for learning individual selectivity. This can be achieved using exclusive execution of sub-plan containing error prone selectivity node.

**Example:** An example for learning sequence of selectivity monitoring is shown in Figure 7.4 with input query location as $q_a$ in ESS. Initial value for selectivity $q_{run}$ is represented by $q_{run}^1$ which is origin of ESS. At this location, Axis-plan ($q_{run}^1$) is found from stored contours location which is \{P_1^1, P_2^1\}. Suppose, $P_1^1$ is chosen for execution and after executing $P_1^1$ database replies with uncompleted execution status and learned selectivity and $q_{run}$ is updated to $q_{run}^2$. Now,
Figure 7.4: Optimized Bouquet Example

Axis-plan\((q_{\text{run}}^2)\) contains only \(P_1^1\), so after executing this plan, \(q_{\text{run}}\) reaches to \(q_{\text{run}}^3\). Axis-plan\((q_{\text{run}}^3)\) is \(\{P_2^2, P_4^2\}\) hence plan \(P_1^2\) and \(P_5^2\) need not to be executed. Suppose, by executing all required plans of contour 2, \(q_{\text{run}}\) is updated to \(q_{\text{run}}^4\) which leads to Axis-plan\((q_{\text{run}}^4)\) to be \(\{P_2^3, P_3^3\}\). This results in eliminating plans \(P_1^3\) and \(P_4^3\) from execution. Finally, query is executed with plan \(P_3^3\).

Using this optimization, four plans \((P_1^2, P_5^2, P_1^3, P_4^3)\) are skipped from execution.

7.3 Implementation Details

These two bouquet algorithms are implemented inside driver program which is developed using Java. This driver program reads preprocessed bouquet data and stores into appropriate data structures. It finds contours and determines plans execution order for basic bouquet and stores contours coordinate location for optimized bouquet.

Driver program also implements bouquet algorithms for running query with extra control of stopping execution after each plan execution. In this feature after a plan execution, state of execution is saved in some variables and execution is stopped until next plan execution instruction comes. Along with these execution features, abstract execution feature is also implemented. In abstract execution, execution process for basic bouquet is virtually shown without executing query on database engine.

The external driver program communicates with database using JDBC (Java Database Connectivity) and utilizes extra implemented features inside database engine. Apart from this, PostgreSQL is modified such that it returns query completion information to driver program.
Chapter 8

QUEST Prototype System

QUEST [4] (QUery Execution without Selectivity esTimation) is prototype implementation of plan bouquet technique. It visually shows bouquet execution process and provides interactivity during execution. QUEST includes modified database system with incorporated features required for bouquet, and implementation of bouquet algorithms that execute query on modified database engine. Currently, PostgreSQL is used for execution wherein all required features are implemented.

8.1 Description

QUEST architecture is shown in Figure 8.1 which comprises of bouquet layer and database engine. Bouquet layer is responsible for finding plans of plan bouquet and executing them on database engine. Inside bouquet layer, steps 1 and 2 find plan bouquet by making optimization calls to database. During step 3, query is executed using plans of bouquet with features like cost bounded execution, selectivity monitoring.

QUEST features range from showing sub-optimality of native optimizer to executing query with bouquet mechanism. These features can be summarized as:

- Showing poor performance of native optimizer by showing native database execution and optimal execution performance.
- Visually showing identification process of plan bouquet.
- Actual and abstract executions of basic and optimized bouquet with visually showing running progress.
8.2 Features Details

In following sections, features or scenarios of QUEST are shown for two-dimensional ESS query based on Query 5 of the TPC-H benchmark [9], operating on 1GB TPC-H database hosted on PostgreSQL engine.

8.2.1 Sub-optimality of Native Optimizer

In first scenario, sub-optimality of native optimizer is shown by presenting selectivity estimation error. Selectivity estimation error causes estimated selectivity $q_e$ to be different from actual selectivity $q_a$. Based on $q_e$ and $q_a$, structures of optimizer chosen plan $P_{oe}$ (optimal at $q_e$) and optimal plan $P_{oa}$ (optimal at $q_a$) are displayed and query is executed with both plans. Based on two execution times, native optimizer’s sub-optimality is shown.

A sample instance of corresponding QUEST interface is shown in Figure 8.2, where:

- An operator level comparison can be observed between native optimizer’s plan $P_{oe}$ and optimal plan $P_{oa}$. In given instance, $P_{oe}$ contains Nested Loop and Hash Joins while $P_{oa}$ contains only Hash Joins, and join order is also different.
- The locations of $q_e$ and $q_a$ are shown in ESS. In given instance, large gap exists between $q_e=(0.25\%, 3.1\%)$ and $q_a=(30.9\%, 26.7\%)$.
- Sub-optimality of native optimizer is shown using bar graph. In this instance, it turns out to be almost 17.
8.2.2 Bouquet Identification

The graphical display for bouquet identification is shown in Figure 8.3. Here, the left picture shows the three-dimensional PIC surface of the native optimizer, characterized by a large number of POSP plans and a steep cost-profile over ESS.

Since the bouquet’s MSO guarantee is a direct function of the POSP cardinality, the dense cost diagram is subjected to anorexic reduction in order to reduce the number of plans to a small number. On this reduced diagram the bouquets distinctive feature of cost-based discretization using geometrically increasing isocost planes is applied – the combined effect of reduction and discretization is presented in the second picture of Figure 8.3.

In the example, the original POSP diagram has 29 plans with a PIC covering the cost range from 1.1E4 to 3.2E5. After anorexic reduction, the plan cardinality goes down to 6 plans. Finally, the PIC is divided using 5 isocost contours, and the POSP plan distribution on these contours is (4, 4, 4, 3, 1).
8.2.3 Bouquet Execution

Now we move to main feature of this implementation – illustrating the bouquet technique’s calibrated sequence of cost bounded partial executions, starting with plans on the cheapest isocost contour, and then systematically working its way through the contours until one of the plans executes the query to completion within its assigned budget. In this scenario, ESS is shown along with all contours using a graph as shown in Figure 8.4. It also provides time line of plan execution, which shows structure of plans those have gone through execution process and shows plan bars of assigned budget height. Basic and optimized bouquet sub-optimality are also presented using bar graph.

Here, query can be executed with basic or optimized bouquet. During execution, other controls are provided to enable pausing the bouquet operation after each partial execution so that the specific progress made through each such execution can be fully assimilated before continuing to the next step. Besides of actual execution, query can be run in abstract execution mode to show sub-optimality of different points in ESS.

Figure 8.3: Bouquet Identification Interface
During bouquet execution:

- The ESS region covered by each partial plan execution, is shadowed with the plan’s color.

- The execution order timeline shows all plans, along with their tree structures – this allows database analysts to carry out offline replays of the plan sequence.

- The contour budgets, which initially appear as white bars of geometrically increasing height in Figure 8.4, and are then filled with blue after the corresponding partial executions (in the figure, after 15 partial executions, plan P6 on Contour 5 completes the query within the assigned budget).

- The sub-optimality of bouquet execution is shown (for the sample query, it is around 3.7).

### 8.2.4 MSO Guarantees

In this scenario, the MSO guarantees offered by the bouquet technique, can be verified by repeated invocations of plan bouquet. Firstly, with regard to the MSO guarantee, $q_a$ needs
to be filled for any desired location in the text box shown in Figure 8.4 (below the isocost contours), and then bouquet algorithm is invoked on query instance to confirm that the sub-optimality incurred is within the apriori stated bound (for the sample query, this MSO bound is 24, which is orders of magnitude lower than the empirically determined MSO of $10^4$ obtained with the native optimizer).

### 8.2.5 Optimized Bouquet

In final scenario of QUEST, an optimized version of the bouquet algorithm is shown that explicitly monitors the selectivities encountered during the partial executions. The monitoring serves to minimize the number of plan executions incurred in crossing contours. With this reduction in overheads, the sub-optimality comes down to just 2.7 in Figure 8.4 for given instance.

![Isocost Contours](image)

Figure 8.5: Optimized Bouquet with Selectivity Monitoring

The impact of this optimization can be observed through a graph that continually tracks the learned selectivity movement in the ESS, as shown in Figure 8.5. Here, the dotted line characterizes the trajectory followed by the bouquet in moving from the origin to the destination $q_a$ location.

### 8.3 Implementation Details

QUEST interface is implemented using Java Swing. It parses input query and checks validity of all relations and attributes by accessing `pg_stats` meta-data relation. QUEST is capable of running queries with two error-prone selectivities. So, it also validates preprocessed bouquet data for running with QUEST. During bouquet identification, cost greedy anorexic reduction [1]
is used for reducing POSP.

In this implementation, graphs and plan trees are drawn using open libraries “JFreeChart” [5] and “JGraph” [6] respectively. 3-D diagrams are generated with Java component library “VisAd” [8] which is used for visualizing numerical data. During bouquet execution, graphs are updated through functionality provided in JFreeChart. QUEST also provides functionality for clearing system cache. This cache clearing function runs system (Linux) command through Java program. Required information for query parsing and showing current database’s sub-optimality is accessed through JDBC from database.
Chapter 9

Conclusions and Future Work

In this work, we discussed necessary techniques for implementing plan bouquet. These techniques include preprocessing, database engine modification, and implementing plan bouquet algorithms. Based on this work, a visual interface QUEST is prepared that shows sub-optimality of native optimizer caused by selectivity estimation problem and confirms empirical validation of plan bouquet mechanism.

Current implementation is prepared for PostgreSQL 8.3. In future, this work can be extended for latest version of PostgreSQL and other commercial database engines. Finding plans of bouquet is done offline from execution process using brute-force technique which is time consuming process. Proposed contours identification algorithm can be incorporated in QUEST which can significantly reduce preprocessing time. Contours obtained from proposed algorithm may contain large number of plans on contours. It will result in large MSO bound guarantee. So numbers of plans can be reduced using modified anorexic reduction to decrease MSO bound such that reduction is valid. Presently, modified PostgreSQL is capable of running bouquet for SPJ (Select-Project-Join) inner join queries which can be extended to run any general query.
Bibliography


