Plan Bouquets: An Exploratory Approach to Robust Query Processing

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Anshuman Dutt

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

August, 2016

DEDICATED TO

My Parents and my wife.

Acknowledgements

I have been extremely fortunate to have Prof. Jayant Haritsa as my research advisor. I am indebted to him for a journey at the end of which I am full of satisfying learnings about research and life and a research project which I will always be proud to have been associated with. He gave me confidence when our work was facing initial rejections and also gave me necessary discipline lessons whenever I lost focus. He never let me compromise me on quality of research and taught me the importance of presentation even if we have good quality work. He has been and will always be my inspiration in many ways.

I also thank Prof. Sudarshan, Prasad Deshpande, Vinayaka Pandit and Sreyash D Kenkre and anonymous reviewers for their feedback and suggestions at various stages during my research. It was a pleasure and learning experience while working with Sumit, Rajmohan, Srinivas, Lohit and Kuntal during collaborative research work. Also, I thank Bruhathi and Dr. Neeldhara Misra for stimulating discussions that later got converted into ideas in the thesis.

Thanks to all the DSL labmates and seniors for making this journey smooth, fun and memorable in many ways. Thanks to Adway, Rakshit for always being there for me; Rafia for everything including the philosophical discussions and life lessons; Adway, Shweta, Manish for giving me impartial outlook on various topics and many many other things.

Acknowledgements

Immense thanks to my parents for accepting my decisions and encouraging me in every possible way. And I thank my wonderful wife Prachi for being my partner in every phase of life and completing me as a person in many ways – this thesis could never have happened without you.

I thank Sushant, Sonal, Durga for being there during my spiritual growth and Gomu sir for being my spiritual guru during my IISc stay.

Finally, I thank all the office staff especially Mrs. Lalitha , Mrs. Suguna, Mrs. Meenakshi and Mr. Shekhar for making all administrative things easy for me and Google (India), VLDB, Department of CSA and Database Systems Lab for supporting my travel to various conferences to present papers on the basis of this thesis.

Abstract

Over the last four decades, relational database systems, with their mathematical basis in first-order logic, have provided a congenial and efficient environment to handle enterprise data during its entire life cycle of generation, storage, maintenance and processing. An organic reason for their pervasive popularity is intrinsic support for *declarative* user queries, wherein the user only specifies the end objectives, and the system takes on the responsibility of identifying the most efficient means, called *"plans"*, to achieve these objectives.

A crucial input to generating efficient query execution plans are the compile-time estimates of the data volumes that are output by the operators implementing the algebraic predicates present in the query. These volume estimates are typically computed using the *"selectivities"* of the predicates. Unfortunately, a pervasive problem encountered in practice is that these selectivities often differ significantly from the values actually encountered during query execution, leading to poor plan choices and grossly inflated response times.

The database research community has spent considerable efforts to address the above challenge, which is of immediate relevance to currently operational systems. The proposed techniques include: (a) Improving estimation accuracy through novel statistical models, sampling and execution-feedback mechanisms; (b) Identifying execution plans that are relatively robust to such errors; and (c) Dynamically changing plans at run-time if estimation errors are detected during the execution of the originally chosen plan. While this rich body of literature features several innovative formulations, the prior techniques all suffer from a systemic limitation – the

inability to provide any *guarantees* on the execution performance.

In this thesis, we materially address this long-standing open problem by developing a radically different query processing strategy that lends itself to attractive guarantees on run-time performance. Specifically, in our approach, the compile-time estimation process is completely eschewed for error-prone selectivities. Instead, from the set of optimal plans in the query's selectivity error space, a limited subset called the "plan bouquet", is selected such that at least one of the bouquet plans is 2-optimal at each location in the space. Then, at run time, an *exploratory* sequence of cost-budgeted executions from the plan bouquet is carried out, eventually finding a plan that executes to completion within its assigned budget. The duration and switching of these executions is controlled by a graded progression of isosurfaces projected onto the optimal performance profile. We prove that this construction provides viable guarantees on the worst-case performance relative to an oracular system that magically possesses accurate apriori knowledge of all selectivities. Moreover, it ensures repeatable execution strategies across different invocations of a query, an extremely desirable feature in industrial settings.

Our second contribution is a suite of techniques that substantively improve on the performance guarantees offered by the basic bouquet algorithm. First, we present an algorithm that skips carefully chosen executions from the basic plan bouquet sequence, leveraging the observation that an expensive execution may provide better coverage as compared to a series of cheaper siblings, thereby reducing the aggregate exploratory overheads. Next, we explore randomized variants with regard to both the sequence of plan executions and the constitution of the plan bouquet, and show that the resulting guarantees are markedly superior, in expectation, to the corresponding worst case values.

From a deployment perspective, the above techniques are appealing since they are completely "black-box", that is, *non-invasive* with regard to the database engine, implementable using only API features that are commonly available in modern systems. As a proof of concept, the bouquet approach has been fully prototyped in QUEST, a Java-based tool that provides

a visual and interactive demonstration of the bouquet identification and execution phases. In similar spirit, we propose an efficient isosurface identification algorithm that avoids exploration of large portions of the error space and drastically reduces the effort involved in bouquet construction.

The plan bouquet approach is ideally suited for "canned" query environments, where the computational investments in bouquet identification are amortized over multiple query invocations. The final contribution of this thesis is extending the advantage of compile-time suboptimality guarantees to ad hoc query environments where the overheads of the off-line bouquet identification may turn out to be impractical. Specifically, we propose a completely revamped bouquet algorithm that constructs the cost-budgeted execution sequence in an "on-the-fly" manner. This is achieved through a "white-box" interaction style with the engine, whereby the plan output cardinalities exposed by the engine are used to compute lower bounds on the error-prone selectivities during plan executions. For this algorithm, the suboptimality guarantees are in the form of a low order polynomial of the number of error-prone selectivities in the query.

The plan bouquet approach has been empirically evaluated on both PostgreSQL and a commercial engine ComOpt, over the TPC-H and TPC-DS benchmark environments. Our experimental results indicate that it delivers orders of magnitude improvements in the worst-case behavior, without impairing the average-case performance, as compared to the native optimizers of these systems. In absolute terms, the worst case suboptimality is upper bounded by 20 across the suite of queries, and the average performance is empirically found to be within a factor of 4 wrt the optimal. Even with the on-the-fly bouquet algorithm, the guarantees are found to be within a factor of 3 as compared to those achievable in the corresponding canned query environment.

Overall, the plan bouquet approach provides novel performance guarantees that open up exciting possibilities for robust query processing.

Publications based on this Thesis

- Anshuman Dutt and Jayant R. Haritsa,
  "Plan Bouquets: A Fragrant Approach to Robust Query Processing",
  In ACM TODS, vol. 41, issue 2, May 2016.

- Anshuman Dutt, Sumit Neelam and Jayant R. Haritsa,
  "QUEST: An Exploratory Approach to Robust Query Processing",
  In VLDB 2014.

- Anshuman Dutt and Jayant Haritsa,
  "Plan Bouquets: Query Processing without Selectivity Estimation",
  In ACM SIGMOD 2014.

- Anshuman Dutt and Jayant R. Haritsa,
  "Query Processing without Estimation",
  Technical Report TR-2014-01, DSL SERC/CSA, IISc, 2014,
  dsl.serc.iisc.ernet.in/publications/report/TR/TR-2014-01.pdf

Contents

Acknowledgements **i**

Abstract **iii**

Publications based on this Thesis **vi**

Contents **vii**

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**

1.1 Cost-based Query Optimization 4

1.2 Selectivity Estimation Errors 5

1.3 Plan Bouquet Approach 7

1.3.1 Performance Characteristics 10

1.4 Summary of Contributions 11

1.4.1 Chapter 4: MSO Bounds and Repeatability 11

1.4.2 Chapter 5: Randomized Bouquet Algorithm 12

1.4.3 Chapter 6: Compile-time Enhancements to Improve MSO Bounds 12

1.4.4 Chapter 7: Efficient Bouquet Identification Mechanism 13

1.4.5 Chapter 8: Plan Bouquet Architecture and Prototype Implementation . . 13
1.4.6 Chapter 9: Empirical Evaluation 14
1.4.7 Chapter 10: MSO Bounds for Ad hoc Queries 14
1.5 Summary 15
1.6 Thesis Organization 15

2 Related Work **17**
2.1 Robustness in Query Processing 17
2.2 Problem Focus 18
2.3 Survey of Existing Techniques 19
2.4 Performance Metric Based Classification 22
2.5 Approach Based Classification 23
2.6 Execution-style Based Classification 25
2.7 Summary 27

3 Problem Framework, Notations and Assumptions **28**
3.1 Query Model 28
3.2 Robustness Model 29
3.2.1 Ancillary Performance Metrics 30
3.3 Notations 31
3.4 Assumptions 32

4 Robustness Bounds using Plan Bouquet Approach **34**
4.1 1D Selectivity Space 34
4.1.1 1D algorithm 34
4.1.2 Performance Analysis 36
4.1.3 Optimality Analysis 37
4.2 Multi-dimensional Selectivity Space 39

4.2.1 Performance Bounds 40

5 Bounds on Maximum Expected Sub-optimality 42

5.1 Randomized Intra-contour Plan Sequence 42

5.2 Randomized Contour Placement 45

5.3 Using the Randomization Strategies in Tandem 48

5.4 Discussion 49

6 Compile-Time Enhancements to Improve Robustness Bounds 51

6.1 Plan Swallowing Enhancement 52

6.2 Execution Covering Enhancement 52

6.2.1 The Cover Relation 54

6.2.2 Motivating Scenario: MSO_g Reduction due to Execution Covering 56

6.2.3 Covering Sequences 57

7 Efficient Identification of Plan Bouquet Sequence 61

7.1 NEXUS: Algorithm for Identifying an Isosurface 62

7.1.1 2D ESS 63

7.1.2 Extension to nD ESS 66

7.1.3 Impact on Bouquet Identification Overheads 69

7.2 Implementing Plan Swallowing 69

7.3 Implementing Execution Covering 70

7.3.1 Covering Sequence Identification Algorithm 70

7.3.2 Efficiently Constructing Hasse Diagram of Executions 73

8 Plan Bouquet Architecture and Prototype Implementation 76

8.1 Bouquet Architecture and Essential API Features 76

8.1.1 Selectivity Injection 76

8.1.2 Abstract Plan Costing and Execution . . . 77
8.1.3 Cost-budgeted Execution . . . 78
8.1.4 Bouquet Driver Layer . . . 78
8.2 QUEST Prototype . . . 78
8.2.1 Sub-optimality of Native Optimizer . . . 79
8.2.2 Bouquet Identification . . . 80
8.2.3 Bouquet Execution . . . 80
8.2.4 MSO Guarantees & Repeatability . . . 82

9 Empirical Evaluation **84**
9.1 Experimental Setup . . . 84
9.2 Compile-time Overheads . . . 87
9.3 Empirical Worst-case Performance (MSO) . . . 87
9.4 Average-case Performance (ASO) . . . 88
9.5 Spatial Distribution of Robustness . . . 88
9.6 Adverse Impact of Bouquet (MH) . . . 89
9.7 Plan Cardinalities . . . 90
9.8 Commercial Database Engine . . . 90
9.9 MSO_g Sensitivity to λ Setting . . . 92

10 MSO Bounds for Ad hoc Queries **93**
10.1 PB_AH for 1D vESS . . . 95
10.1.1 Cost-Budgeted Planning (CBP) . . . 95
10.1.2 Characteristic of 1D Executions . . . 97
10.2 PB_AH for 2D vESS . . . 98
10.2.1 Intermediate Queries . . . 99
10.2.2 Decomposition into Subproblems . . . 100

10.2.3 Solving 1D Subproblems Using Repeated Invocations of CBP 101
10.2.4 Implications of 1D Executions 102
10.2.5 Number of 1D Executions to Cross an Isosurface 103
10.3 $\mathsf{PB}_{\mathsf{AH}}$ for Multi-D vESS 104
10.3.1 Implications of 1D Executions 104
10.3.2 $\mathsf{MSO}_{\mathsf{g}}$ Analysis for Generic Multi-D vESS 107
10.4 Performance Results 109
10.4.1 MSO bounds 109
10.4.2 Other empirical observations 109

11 Discussion **111**
11.1 Revisiting Assumptions 111
11.1.1 Selectivity Independence Assumption 111
11.1.2 Perfect Cost Model 115
11.1.3 No known Selectivity Bounds and Lack of Absolute Metric 116
11.2 Critique of Bouquet Approach 118
11.2.1 Deployments Aspects 119

12 Conclusions and Future Directions **120**
12.1 Conclusions 120
12.1.1 Relevance in Non-relational Systems 121
12.2 Future Directions 122

Bibliography **126**
12.A Appendix 137
12.A.1 Query Text (based on benchmark queries) 137

List of Figures

1.1 Example SQL Query (EQ) . . . 1
1.2 Sample Execution Plan for EQ . . . 2
1.3 Traditional RDBMS architecture . . . 4
1.4 POSP plans on p_retailprice dimension . . . 6
1.5 POSP performance (log-log scale) . . . 8
1.6 Bouquet performance (log-log scale) . . . 10

2.1 Classification on the basis of approach to handle errors . . . 23

4.1 1D selectivity space . . . 35
4.2 2D Selectivity Space: (a) Isocost Contours (b) Space coverage by plans on IC_k . . . 40

5.1 Worst-case and best-case (intra-contour) plan sequences for q_a . . . 43
5.2 Worst-case and best-case contour placements for q_a . . . 46

6.1 E_7 can complete all locations in $R(E_3)$ with cost(IC_2) = 2 * cost(IC_1) . . . 53
6.2 Example bouquet sequence . . . 55
6.3 (a) Hasse diagram (b) Execution cost and sub-optimality analysis . . . 55
6.4 Using cover $E_{28} \rightarrow \{E_{20}, E_{21}, E_{22}\}$ improves $\mathsf{MSO_g}$ from 24.25 to 22.25 . . . 57
6.5 (a) Execution covering sequence (b) Modified space coverage . . . 59

7.1 Finding seed location in 2D ESS . . . 65

7.2 Intermediate Contour Exploration in 2D ESS . . . 65
7.3 Completely Explored Contour in 2D ESS . . . 66
7.4 Example Contour Exploration in 3D ESS (Case 2b) . . . 67
7.5 Example Contour Exploration in 3D ESS (Case 1) . . . 68
7.6 Example Contour Exploration in 3D ESS (Case 2a) . . . 68
7.7 Adapting Red-Blue Weighted Domination for solving *minimize*$(w(CS^4))$ . . . 70
7.8 Solution for *minimize*$(w(CS^4))$ using red-blue domination . . . 72
7.9 MAX_i and MIN_i locations for executions on contours IC_1 to IC_3 . . . 74

8.1 Architecture of Bouquet Mechanism . . . 77
8.2 Sub-optimality of Native Optimizer . . . 79
8.3 Bouquet Identification Interface . . . 81
8.4 Bouquet Execution Interface . . . 82

9.1 Empirical MSO Performance . . . 88
9.2 ASO Performance . . . 89
9.3 Distribution of enhanced Robustness (5D_DS_Q19) . . . 89
9.4 MaxHarm performance . . . 90
9.5 Bouquet Cardinality . . . 91
9.6 Commercial Engine Performance (log-scale) . . . 91
9.7 MSO_g vs Reduction-parameter (λ) . . . 92

10.1 Cost-budgeted planning (CBP) . . . 96
10.2 SLPs for $P1$ and $P2$ for costs $C1$ and $C2$ . . . 98
10.3 Example TPC-H Query EQ2 . . . 99
10.4 Join Graph for EQ2 . . . 99
10.5 Intermediate relations for the example query EQ2 . . . 100
10.6 Intermediate query for PLO . . . 100

10.7 Intermediate relations and their cardinality expressions 101
10.8 Execution with maximum SLP using repeated invocations of CBP 102
10.9 Bouquet Algorithm for Adhoc queries 105
10.10MSO_g increase with dimensions 109

11.1 Cardinality expressions with and without join-predicate independence assumption 113
11.2 Utilizing lower bound (LB) and upper bound (LB) on selectivities 116
11.3 Variation of MSO_g and $\frac{TAC}{C_{max}}$ with cost-ratio r 117

12.1 3D_H_Q5 (Based on TPC-H Query 5) 137
12.2 3D_H_Q5^b (Based on TPC-H Query 5) 138
12.3 3D_H_Q7 (Based on TPC-H Query 7) 138
12.4 5D_H_Q7 (Based on TPC-H Query 7) 139
12.5 4D_H_Q8 (Based on TPC-H Query 8) 139
12.6 4D_H_Q8^b (Based on TPC-H Query 8) 140
12.7 4D_DS_Q7 (Based on TPC-DS Query 7) 141
12.8 3D_DS_Q15 (Based on TPC-DS Query 15) 141
12.9 5D_DS_Q19 (Based on TPC-DS Query 19) 142
12.104D_DS_Q26 (Based on TPC-DS Query 26) 142
12.114D_DS_Q91 (Based on TPC-DS Query 91) 143
12.123D_DS_Q96 (Based on TPC-DS Query 96) 143

List of Tables

3.1 Reference table for Notations 32

5.1 Performance of randomized variants of the bouquet algorithm 50

6.1 Effect of Anorexic Reduction [$\lambda = 20\%$] on Robustness Guarantees 53

6.2 Reference table for Notations for Execution Covering 54

6.3 Effect of Execution Covering on Robustness Guarantees 60

7.1 Reference table for Notations for NEXUS algorithm 63

9.1 Query workload specifications 85

10.1 Guarantees in ad hoc query environment 110

11.1 MSO guarantees without join-predicate independence assumption 114

Chapter 1

Introduction

It has been more than four decades since the relational model [31] of data representation enabled a significant step forward in database query processing by removing the dependence between application programs and underlying data representation. As a result, modern database systems provide a *declarative* query interface, typically in the form of SQL, that allow the user to specify *what* information from the database is needed without having to specify *how* to retrieve it from the data and compute query results. In Figure 1.1, we show an example SQL query **EQ** over TPC-H schema that enumerates the orders for cheap parts.

```
SELECT *
FROM part, lineitem, orders
WHERE p_partkey = l_partkey and
      o_orderkey = l_orderkey and
      p_retailprice < 1000
```

Figure 1.1: Example SQL Query (EQ)

To support efficient data access in these systems, the task of identifying the most *time-efficient* procedural equivalent of the input query, termed as 'execution plan', is performed by a module called the *query optimizer*. Soon after the proposal of relational model, System R [82] developed *cost-based* query optimization wherein alternative execution plans are compared on the basis of their cost, i.e. expected time to complete execution, and the minimum cost choice

among them is picked for execution. Since then, cost-based query optimization has served as a template for query optimizer design.

An execution plan (or just plan) is a sequence of relational operators that produce the query result by evaluating the predicates specified in the query. For instance, a sample plan for query EQ is shown in Figure 1.2 where the predicates in the query are evaluated using relational scan and binary join operators with different algorithmic choices, e.g. Index Scan and Sort-Merge Join, respectively. The total time taken by a plan to complete query execution depends on the *selectivity* of the query predicates, i.e. fraction (or percentage) of data tuples satisfying the predicate, and the physical implementation of the operators.

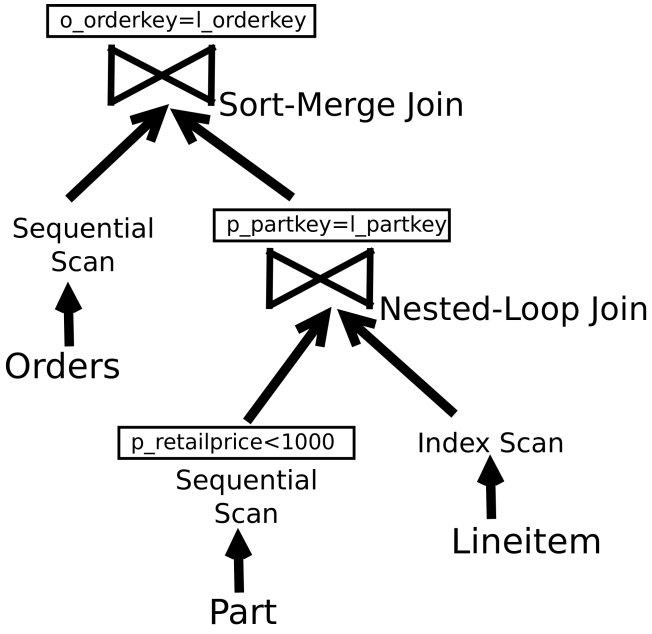


Figure 1.2: Sample Execution Plan for EQ

To compare across various execution choices, query optimizers employ two abstract models: (1) operator output selectivity (normalized cardinality) estimation model, and (2) operator execution cost estimation model. Clearly, the ability of a cost-based optimizer to identify the ideal execution plan is dependent on the quality of these models. Specifically, the quality of selectivity estimation model is a function of its ability to capture the distributions and

correlations present in the data, while that of cost model depends on how well it captures the behavior of the underlying hardware and physical implementations of the operators.

While there has been a plethora of research proposals to improve the quality of these models, query optimization has largely remained a "black art", as highlighted by the following comments from the leading academic and industry experts:

Prof. David Dewitt (Univ. of Wisconsin Madison, Microsoft Jim Gray Lab) [86]: *Query optimizers do a terrible job of producing reliable, good plans (for complex queries) without a lot of hand tuning.*

Dr. Surajit Chaudhuri (Microsoft Research) [75]: *Almost all of us who have worked on query optimization find the current state of the art unsatisfactory with known big gaps in the technology.*

Dr. Guy Lohman (IBM Research) [62]: *With such errors (in cardinality estimation), the wonder isn't "Why did the optimizer pick a bad plan?" Rather, the wonder is "Why would the optimizer ever pick a decent plan?"*

In this thesis, motivated by the above comments on this long standing issue, we present a radically new approach to database query processing that lends itself to attractive guarantees on run-time execution performance, regardless of the actual selectivities. Based on this approach, we propose techniques that provide performance guarantees *orders of magnitude* better than the worst-case performance of native engines, for both canned as well as ad hoc query environments. Moreover, these techniques can be deployed in a *non-invasive* manner with regard to the existing database engine.

1.1 Cost-based Query Optimization

With the aim of identifying the ideal execution plan, cost-based query optimizers estimate a host of *selectivities* corresponding to the predicates in the query. For example, for simple SPJ[1] query **EQ**, the optimizer estimates the selectivities of a selection predicate ($p\_retailprice < 1000$) and two join predicates ($part \bowtie lineitem$, $orders \bowtie lineitem$). The selectivities are estimated using statistical metadata (histograms, distinct counts, etc), and assumptions like attribute value independence, join containment assumption, etc [25]. These selectivity estimates serve as primary inputs to a cost model that compares various execution plan choices to determine the cost-optimal execution plan. The chosen execution plan, which is a tree of unary and binary relational algebra operators (select, project, join, etc) instantiated with physical operator algorithms, is then fed to the query executor module. Finally, the executor module follows the operator sequence to interact with the underlying data source and produce the query result tuples. These steps of query processing in relational databases are visualized in Figure 1.3.

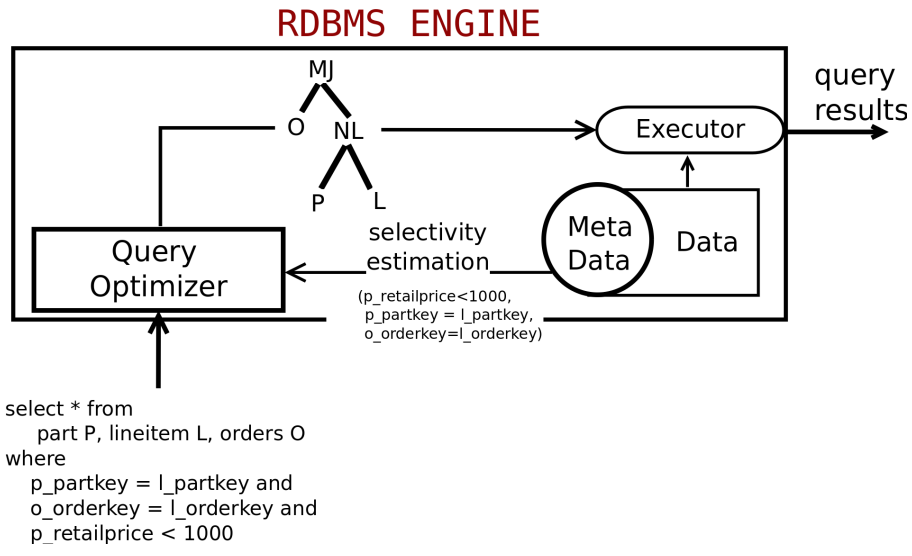


Figure 1.3: Traditional RDBMS architecture

[1] An SPJ query essentially represents a single SQL SELECT-FROM-WHERE block with no grouping or aggregation or subqueries. Further, an SPJ query with only conjunctive predicates in the WHERE block is called a conjunctive SQL query.

The cost-based query optimizers, pioneered by System R [82] and refined extensively by later research, deliver satisfactory performance whenever the queries are simple, modeling assumptions are valid, and the meta-data information is fresh and sufficient for selectivity estimation of predicates. But with the growth in query complexity, the challenge of identifying the optimal execution plan has increased manifold [51]. Also, the 'suboptimality' due to mistaken plan choice, i.e. the performance ratio wrt to the ideal plan choice, increase with the growth in data scale and skew. The current situation is that for complex OLAP queries over large databases, the selectivity estimates are often significantly in error with respect to the actual values subsequently encountered during query execution, leading to grossly inflated execution times. Also, it is widely accepted that the suboptimality impact of selectivity estimation errors is significantly larger as compared to that of the cost modeling errors [62, 58, 87].

In this thesis, we first study individual impact of selectivitity estimation errors by assuming the cost model to be *perfect* – that is, we use only *optimizer costs* in the evaluations. While this assumption is certainly not valid in practice, improving the cost model quality is, in principle, an orthogonal problem to that of selectivity estimation. Later, at the end of the thesis in Chapter 11, we revisit this assumption and analyze the impact of having an erroneous cost model on the results of this thesis.

1.2 Selectivity Estimation Errors

Selectivity estimation errors, which can even be in *orders of magnitude* in real database environments [65, 62], arise due to a variety of well-documented reasons [84, 62, 85], including outdated statistics, coarse summaries, attribute-value independence (AVI) assumptions, complex user-defined predicates, and error propagation in the query execution operator tree [51]. Moreover, in environments such as ETL workflows, the statistics may actually be *unavailable* due to data source constraints, forcing the optimizer to resort to "magic numbers" for the values (e.g. 1/10 for equality selections [82]).

To analyze the impact of estimation errors, consider a restricted 1D version of the query EQ wherein only the selection predicate p_retailprice $<$ 1000 is assumed to be error-prone. Here, we find that the plan choices of the query optimizer change as a function of the selectivity of the predicate p_retailprice $<$ 1000. Specifically, through repeated invocations of the optimizer with increasing selectivity value for the selection predicate[1], we identify "parametric optimal set of plans" (POSP) and their optimality ranges across the entire selectivity range of the predicate, i.e. plans P1 through P5, whose structures and optimality ranges are shown in Figure 1.4. Further, each plan is annotated with the selectivity range over which it is optimal – for instance, plan P3 is optimal in the (1.0%, 7.5%] interval. (In Figure 1.4, P = Part, L = Lineitem, O = Orders, NL = Nested Loops Join, MJ = Sort Merge Join, and HJ = Hash Join).

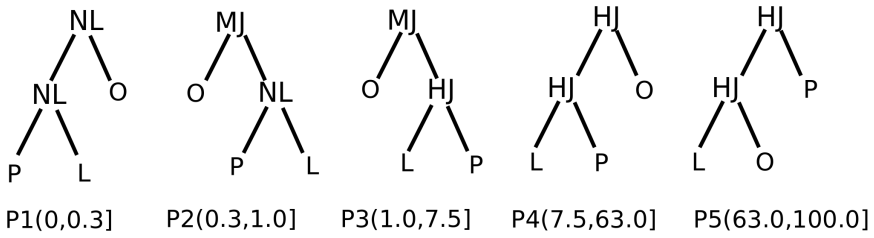


Figure 1.4: POSP plans on p_retailprice dimension

Next, observing the cost behavior of these plans (in Figure 1.5), it is found that if the actual selectivity is close to 90% but underestimated to be in the range $(0, 0.3\%]$ by the optimizer, then EQ is executed using plan P1 resulting in 20 times sub-optimal performance. On the other hand, if the actual selectivity is around 0.1% and overestimated to be 90%, then the performance suboptimality is close to *two orders* of magnitude. Even for the simple base predicate in EQ, such errors can actually happen due to staleness of the statistical information for a highly skewed data distribution. Even if the statistics were fresh, the selectivity estimation module of most engines would easily make similar large errors if the predicate happen to be written as $\log_{10}$(p_retailprice) $<$ 3.

A considerable body of literature exists on proposals to tackle the issue of erroneous selec-

[1]This is achieved by leveraging 'selectivity injection' API, as described later in Chapter 8.

tivity estimation. For instance, techniques for improving the *statistical quality* of the meta-data include improved summary structures [7, 69], sampling [61, 74], feedback-based adjustments [84, 7], and on-the-fly re-optimization of queries [52, 12, 72]. A complementary approach is to identify *robust plans* that are relatively less sensitive to estimation errors [29, 10, 12, 47, 21]. While these prior techniques provide novel and innovative formulations, they are limited in their scope and performance, as explained in detail in Chapter 2 – a primary drawback being lack of performance guarantees.

1.3 Plan Bouquet Approach

In this thesis, we investigate a conceptually new approach to database query processing, wherein the compile-time estimation process is completely eschewed for error-prone selectivities. Instead, these selectivities are systematically *discovered* at run-time through a calibrated sequence of cost-budgeted plan executions. That is, we attempt to side-step the selectivity estimation problem, rather than address it head-on, by adopting a *"seeing is believing"* perspective on these values.

1D Example We introduce the new approach through the restricted 1D version of the EQ example query, as discussed earlier. The process starts with repeated invocations of the optimizer to identify the "parametric optimal set of plans" (POSP) that cover the entire selectivity range of the predicate. A sample outcome of this process is already shown in Figure 1.4 and the optimizer-computed costs of these POSP plans over the selectivity range are shown (on a log-log scale) in Figure 1.5. In this figure, we define "POSP infimum curve" (**PIC**), as the trajectory of the minimum cost choice from among the POSP plans – this curve represents the ideal performance.

The next step, which is a distinctive feature of our approach, is to *discretize* the PIC by projecting a graded progression of *isocost* (IC) steps onto the curve. For example, in Figure 1.5, the dotted horizontal lines represent a geometric progression of isocost steps, IC1 through IC7,

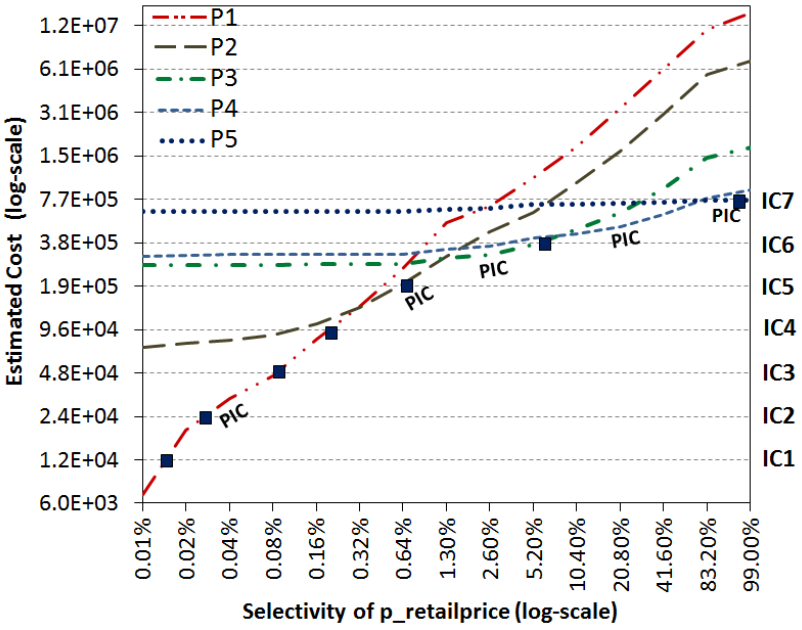


Figure 1.5: POSP performance (log-log scale)

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 1.5, the intersection of IC5 with the PIC corresponds to a selectivity of 0.65% with associated POSP plan P2. We term the subset of POSP plans that are associated with the intersections as the "**plan bouquet**" for the given query – in Figure 1.5, the bouquet consists of {P1, P2, P3, P5}.

The above exercise is carried out at query compilation time. Subsequently, at run-time, the correct query selectivities are *implicitly* discovered through a exploratory sequence of *cost-budgeted* executions of bouquet plans. Specifically, beginning with the cheapest cost step, we iteratively execute the bouquet plan assigned to each step until either:

1. The partial execution overheads exceed the step's cost value – in this case, we know that the actual selectivity location lies beyond the current step, motivating a switch to the next step in the sequence; or
2. The current plan completes execution within the budget – in this case, we know that the

actual selectivity location has been reached, and a plan that is at least *2-optimal* wrt the ideal choice, was used for the final execution.

Example To make the above process concrete, consider the case where the selectivity of p_retailprice is 5%. Here, we begin by partially executing plan P1 until the execution overheads reach IC1 (1.2E4 | 0.015%). Then, we extend our cost horizon to IC2, and continue executing P1 until the overheads reach IC2 (2.4E4| 0.03%), and so on until the overheads reach IC4 (9.6E4 | 0.2%). At this juncture, there is a change of plan to P2 as we look ahead to IC5 (1.9E5 | 0.65%), and during this switching all the intermediate results (if any) produced thus far by P1 are *discarded*. The new plan P2 is executed until the associated overhead limit (1.9E5) is reached. The cost horizon is now extended to IC6 (3.8E5 | 6.5%), in the process discarding P2's intermediate results and executing P3 instead. The execution in this case will complete before the cost limit is reached since the actual location, 5%, is less than the selectivity limit of IC6. Viewed in toto, the net suboptimality turns out to be 1.78 since the exploratory overheads are 0.78 times the optimal cost, and the optimal plan itself was (coincidentally) employed for the final execution.

Extension to Multiple Dimensions When the above 1D approach is generalized to a 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 2D case, the IC steps are horizontal planes cutting through a hollow three-dimensional PIC surface, typically resulting in hyperbolic intersection contours featuring a multitude of plans covering disjoint segments of the contours – an instance of this scenario is shown in Figure 4.2.

Notwithstanding these changes, the basic mechanics of the bouquet algorithm remain virtually identical. The primary difference is that we jump from one isosurface to the next only after it is determined that *none* of the bouquet plans present on the current isosurface can completely execute the given query within the associated cost budget.

1.3.1 Performance Characteristics

At first glance, the plan bouquet approach, as described above, may appear to be utterly absurd and self-defeating because: (a) At compile-time, considerable preprocessing may be required to identify the POSP plan set and the associated PIC; and (b) At run-time, the overheads may be hugely expensive since there are multiple plan executions for a single query – in the worst scenario, as many plans as are present in the bouquet!

However, we will attempt to make the case in the remainder of this thesis, that it is indeed possible, through careful design, to have *plan bouquets efficiently provide robustness profiles that are markedly superior to the native optimizer's profile*. Specifically, we define robustness to be "the worst-case suboptimality in plan performance that can arise due to selectivity errors", denoted as **MSO** (maximum sub-optimality)[1]. With regard to this MSO metric, the bouquet mechanism delivers substantial improvements over current optimizers. Moreover, it does so while providing comparable or improved average-case performance.

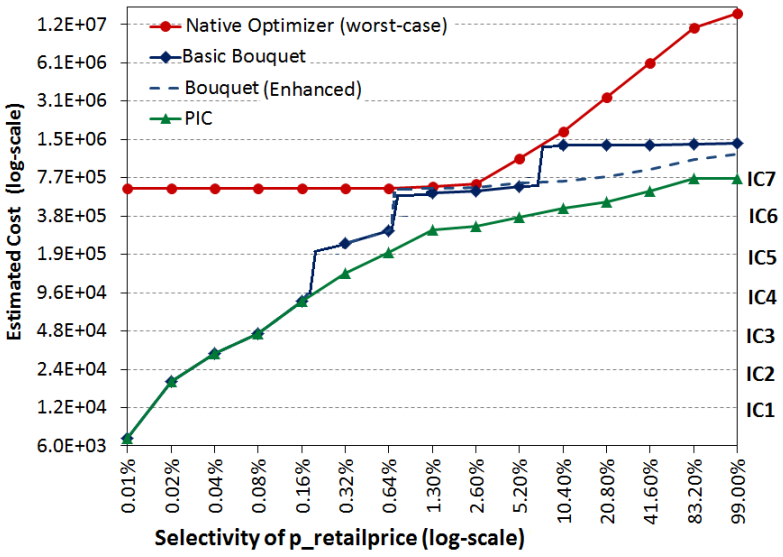


Figure 1.6: Bouquet performance (log-log scale)

For instance, the runtime performance of the bouquet technique on EQ is profiled in Fig-

[1]Precise definition given in Chapter 3.

ure 1.6 (dark blue curve). We observe that its performance is much closer to the PIC (dark green) as compared to the worst case profile for the native optimizer (dark red), which is comprised of the supremum of the individual plan profiles. In fact, the MSO for the bouquet is only 3.6 (at 6.5%), whereas the native optimizer suffers a suboptimality of around 100 when P5 (which is optimal for large selectivities) is mistakenly chosen to execute a query with a small selectivity of 0.01%. The *average* suboptimality of the bouquet, computed over all possible errors, is 2.4, somewhat worse than the 1.8 obtained with the native optimizer. However, when the enhancements described later in this thesis are incorporated, the enhanced bouquet's performance (dashed blue) improves to 3.1 (worst case) and 1.7 (average case), thereby dominating the native optimizer on both metrics.

1.4 Summary of Contributions

In a nutshell, this thesis presents the first-ever set of techniques in OLAP query processing that provide guarantees on execution performance at query compilation time. The guarantees are in the form of upper bounds on MSO independent of the actual selectivities of the query predicates. They are achieved by substituting the selectivity estimation module with a selectivity discovery framework for error-prone predicates. It is to be emphasized that the techniques are *non-invasive* with regard to the database engine and can be successfully implemented using only API features (e.g. *selectivity injection*, *abstract plan costing*, etc.) that have already found expression in modern industrial DB engines – as explained later in Chapter 8. The individual contributions of the thesis are summarized below, on a chapter by chapter basis.

1.4.1 Chapter 4: MSO Bounds and Repeatability

The primary contribution of the thesis is the *novel cost-budgeted execution sequence* that is constructed using the cost-based discretization of the PIC leading to *guaranteed* upper bounds on MSO. For instance, we prove that the cost-doubling strategy used in the 1D example results in an *MSO upper-bound of 4* – this bound is inclusive of all exploratory overheads incurred by

the partial executions, and is *irrespective* of the query's actual selectivity. In fact, we can go further to show that 4 is the best competitive factor achievable by *any* deterministic algorithm. For the multi-dimensional case, the MSO bound becomes 4 times the bouquet cardinality (more accurately, the plan cardinality of the densest isosurface).

Apart from improving robustness, there is another major benefit of the bouquet mechanism: On a given database, the execution strategy for a particular query instance, i.e. the sequence of plan executions, is *repeatable* across different invocations of the query instance – this is in marked contrast to prior approaches wherein plan choices are influenced by the current state of the database statistics and the query construction. Such stability of performance is especially important for industrial applications, where considerable value is attributed to reproducible performance characteristics [10].

1.4.2 Chapter 5: Randomized Bouquet Algorithm

In addition to the deterministic algorithm, we also explore randomization opportunities in the plan bouquet approach. Specifically, we propose randomization of *intra-surface plan sequence* and *isosurface placement*, such that the guarantees on maximum expected suboptimality are markedly superior to their worst-case counterparts. In fact, we show that both randomizations can also be used in tandem to achieve even better bounds on maximum *expected* suboptimality.

1.4.3 Chapter 6: Compile-time Enhancements to Improve MSO Bounds

As discussed above, the MSO bound for any query with multiple error-prone predicates is a function of the plan-densities of the isosurfaces in its selectivity error space. In fact, it was empirically found that the plan densities for complex benchmark OLAP queries can be in the range of hundreds. To handle these large values of isosurface plan-densities leading to high absolute value of guarantees, we present a suite of compile-time enhancements to the basic plan bouquet algorithm that result in materially improved suboptimality guarantees. In this

direction, we first utilize the concept of *plan-swallowing* [46] to reduce the maximum isosurface plan-density by reducing the set of optimal plans in the entire selectivity error space. Next, we introduce the concept of *execution-covering*, where we utilize the observation that multiple executions with smaller cost-budgets can be skipped, without any adverse impact on the MSO guarantee, if their *collective role* can be played by a carefully identified execution with a larger cost-budget.

1.4.4 Chapter 7: Efficient Bouquet Identification Mechanism

The bouquet construction requires to identify *only isosurfaces* with geometrically increasing cost values. As a result, large sections of the selectivity error space do not directly contribute to the bouquet construction. Based on this observation, we propose an algorithm that traces only the locations along the isosurfaces and avoids the exploration of unnecessary portions of the space. In principle, it is possible that the plan-swallowing enhancement does not remain as effective as shown in [46] due to restricted knowledge of POSP. In this regard, we empirically found that the benefits of plan-swallowing at *intra-surface* level continues to be comparable to the benefits achieved with complete information about the selectivity error space.

1.4.5 Chapter 8: Plan Bouquet Architecture and Prototype Implementation

This is followed by the generic architecture of the bouquet approach and the details regarding the required API features to support *non-invasive* implementation of the technique with regard to the database engine. Specifically, there is a external 'Bouquet Driver' program that treats the database engine as a *black-box*. At compile-time, it constructs an execution-sequence using only calls to the query optimizer module and at run-time, it requires an executor module that responds to any cost-budgeted execution by only notifying its completion-status. Finally we discuss about QUEST, a Java-based prototype implementation of the bouquet technique that provides a visual demonstration of bouquet identification as well as execution phase.

1.4.6 Chapter 9: Empirical Evaluation

In order to empirically validate its utility, we have evaluated the bouquet approach on both PostgreSQL and a popular commercial database ComOpt. Our experiments utilize a rich set of complex decision support queries sourced from the TPC-H and TPC-DS benchmarks. The query workload includes selectivity spaces with as many as *five* error-prone dimensions, thereby capturing environments that are extremely challenging from a robustness perspective. Our performance results indicate that the bouquet approach typically provides *orders of magnitude* improvements, as compared to the optimizer's native choices. As a case in point, for Query 19 of the TPC-DS benchmark with 5 error prone join selectivities, the MSO plummeted from about 10^6 to just 10! The potency of the approach is also indicated by its providing an MSO guarantee of less than 20 over our entire query workload, while the average suboptimality was typically within a factor of 4 wrt the optimal.

What is even more gratifying is that the above performance profiles are *conservative* since we assume that at every plan switch, *all* previous intermediate results are completely thrown away – in practice, it is conceivable that some of these prior results could be retained and reused in the execution of a future plan.

1.4.7 Chapter 10: MSO Bounds for Ad hoc Queries

In this chapter, we address those query scenarios where the compile-time bouquet construction phase may not be practical, e.g. ad hoc query environment. Specifically, we show that the compile-time guarantees of the bouquet technique can be extended to such environments as well, by relaxing only the black-box interaction assumption to *white-box* engagement, i.e., the engine also reports the output cardinality for individual cost-budgeted executions which is then used to compute lower bounds on error-prone selectivities.

For this purpose, we propose a completely revamped algorithm that enables "on-the-fly" construction of the cost-budgeted execution sequence and provides compile-time suboptimality

guarantees in the form of a low order polynomial in the number of error-prone selectivities of the query. With this algorithm, the MSO guarantees for our suite of queries are empirically found to be within a factor of 3 wrt the guarantees achievable with the offline version.

1.5 Summary

In closing, we wish to highlight that from a deployment perspective, the bouquet technique is intended to complementarily co-exist with the classical optimizer setup, and not to replace it. It is left to the user or DBA to make the choice of which system to use for a specific query instance – essential factors that are likely to influence this choice are discussed in Chapter 11. Here, we also discuss the limitations of the bouquet approach and revisit some of the assumptions made in the thesis with initial ideas in the direction of their relaxation.

Overall, the bouquet approach provides novel performance guarantees that open up new possibilities for robust query processing.

1.6 Thesis Organization

The remainder of the thesis is organized as follows: We start with reviewing the related literature in Chapter 2 followed by a precise description of the robust query processing problem, along with the underlying assumptions and notations in Chapter 3. Theoretical bounds on the MSO provided by the bouquet technique are presented in Chapter 4 followed by randomized variants in Chapter 5 with the corresponding bounds on maximum expected suboptimality. Then, Chapter 6 discusses the compile-time enhancements that result in significantly stronger MSO guarantees. Next, we describe an efficient mechanism to achieve pragmatic overheads for bouquet identification in Chapter 7, followed by other implementation details of the plan bouquet architecture and prototype in Chapter 8. Further, the experimental framework and performance results for the black-box techniques are reported in Chapter 9. In the end, we present the white-box technique to achieve performance guarantees for ad hoc queries in Chap-

ter 10. Finally, we present ideas to relax the optional simplifying assumptions, critique the bouquet technique in Chapter 11 and conclude in Chapter 12.

Chapter 2

Related Work

In this chapter, we start with a brief discussion on the concept of robustness, in particular wrt database query processing and possible reasons that cause lack of robustness followed by the details regarding the focus area of this thesis, i.e. selectivity estimation errors. Next, we provide a quick recap of the existing literature that handles the erroneous estimates in different ways. Finally, while there are many possible ways of classifying the existing techniques, we put the plan bouquet approach in perspective by using classification on the basis of: (a) performance metric, (b) approach (reactive, proactive or non-traditional) and (c) execution style.

2.1 Robustness in Query Processing

While in generic context of data management, robustness includes redundancy, disaster preparedness and recovery from physical disk failure, etc. Robust query processing, is specifically about the performance predictability and the ability to avoid sudden disruptions in query execution performance [44]. The possible reasons for disruption in execution performance are:

- error(s) in estimated selectivities,
- changes in physical database design or available materialized views,
- changes in data source characteristics including partitioning or network disruption or streaming characteristics, and

- unstable execution environment due to changes in resource availability, fluctuating workloads or conflicts in concurrency control.

But, there have been a series of research studies, panels and seminars in the past decade which argue that inspite of all the above proposals, database query processing and its robustness is still an unsolved and highly relevant issue [41, 44, 58, 38, 43, 81, 69, 20, 25].

2.2 Problem Focus

Our focus in this work is primarily the performance disruptions due to errors in selectivity calculations. It has been well accepted that the impact of these errors on execution performance is significant enough to receive special attention – as pointed out by the following recent statement from industry expert Dr. Guy Lohman [62]:

"The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. Everything in cost estimation depends upon how many rows will be processed, so the entire cost model is predicated upon the cardinality model. In my experience, the cost model may introduce errors of at most 30% for a given cardinality, but the cardinality model can quite easily introduce errors of ***many orders of magnitude****!"*

Query processing environment To clearly describe our focus we now describe the query processing environment assumed in this thesis,

- static data source,
- physical design and available views are fixed and adaptive indexing is not in action,
- query execution cost is not affected by the run-time issues like resources availability etc.

  Later in the thesis (Chapter 11), we analyze the impact of run-time conditions in terms of maximum deviation from estimation cost/time.

2.3 Survey of Existing Techniques

Histograms and other statistical structures Soon after the pioneering work of System-R [82], it was realized that selectivities estimated based on the uniform distribution assumption will mostly mislead the query optimizer due to the skew in data. Thus, to capture the data distribution without requiring lot of space, [55, 76, 27] proposed use of histograms followed by use of frequent values as complementary statistics [64] to handle extreme skew. These methods work remarkably well for computing selectivities of single attribute predicates and are used by most database systems till date in various forms. Recently, the research on histograms has been revived by proposals of histograms that minimize *q-error* metric [69, 68] and heterogeneous histograms [53], i.e., using different types buckets in the same histogram.

While histograms support reasonable selectivity estimates for single attribute predicates, there are other challenging assumptions [28] that affect the process of plan comparison in the query optimizer. Among these, the most critical assumption is that of attribute value independence (AVI) which ignores attribute correlations and is responsible for huge errors in selectivity estimates for multiple-attribute predicates and joins [51]. To capture such multi-dimensional information, multi-dimensional histograms were proposed in [70] and developed further by [78, 56, 33]. But such histograms were not well accepted in industry due to the associated storage and maintenance overheads. These overheads can be controlled to some extent by utilizing self-tuning histograms [7, 19, 60] which can be updated using the feedback information from the queries. A survey of other developments in histogram techniques can be found in [50].

Apart from histograms, other ways of summarizing data distributions include use of wavelet transform [66, 39] and discrete cosine transform (DCT) [57] that are much better in terms of space efficiency. Recently, there have been proposals to capture the data distribution information with probabilistic models [40], graph-based techniques [83], graphical models [85]

and kernel density estimators [48]. All these innovative techniques have the ability to represent statistical information with very less space requirement but face creation and maintenance challenges with increase in the number of attributes in the query predicates.

Random sampling Random sampling is another promising line of research that has been explored as a complementary technique to that of histograms. It is used to reduce overheads in creation and maintenance of histograms [70, 23] and to produce better estimates in difficult scenarios, e.g., multiple predicate combinations, as it scales better than histograms with increase in the number of attributes in the predicates. Also, it has been shown that with adaptive sampling [61], the number of samples required for a given accuracy is independent of the data scale. Although, given sufficient number of samples, it can produce more accurate estimates than histograms but it requires frequent access to relations on the slow secondary storage and hence does not satisfy the requirement of small query optimization time. Further for the case of join estimates, while it has been shown that trivial random sampling may need prohibitively large number of samples [22], there has been proposals to achieve practical solutions with the help of extra frequency statistics or index [22, 88, 36, 74].

Inter-query feedback cache Another interesting research direction is where information relevant to the queries is learned from the feedback received from the workload queries itself [26, 84] and utilized to improve estimates for future queries. While, such technique is inexpensive to implement, it does not ensure significant help as learned information may not be relevant across workload queries. Still, this direction has received significant attention and developed further to learn statistics over query expressions [18, 17, 35] because of its ease of implementation and lack of any side-effect. Recent proposals also presented proactive ways [24] of learning extra information from query plans by introducing appropriate modifications in their structure.

Robust plans All the above proposals tried to improve the selectivity estimates for the query predicates and choose the ideal execution plan using the estimates. One more approach has been to expect the estimates to be erroneous and modify the plan selection process such that, the

criteria is not cost-optimality for a given estimated selectivity but its sensitivity to estimation errors or performance over a range of selectivities. The examples for such approaches include: plan with least expected cost [30, 29], plan chosen using robust cardinality estimates [10], plan that is resistant to unbounded estimation errors [47] and plan with minimum variance across selectivity error space [21].

Another series of attempts tried to construct at optimization time, plan with *specialized* operator, such that they are better equipped to handle the estimation errors at execution-time, they include complex operators with ability to switch among them [32] and more recently smooth operators [16, 42].

Re-optimization/Intra-query execution feedback The research directions summarized above always commit to single plan during optimization time and use to during the entire execution process, and hence they are limited in their capability to handle unknown scenarios at execution time. To overcome this limitation, techniques were proposed to detect sub-optimality during execution and going back to the optimizer with the additional information gathered during the partial execution to pick another plan [52, 65, 12]. While these techniques certainly perform better than initial plan choice but it is not always possible to detect sub-optimality and it may require multiple iterations leading to increased total overheads.

There have also been attempts that use the execution feedback to *reorder* the operators in the optimizer choice plan [59, 67] in the hope to achieve an ordering with lesser cost. As an extreme version of such approaches, there have been attempts that forgo the concept of a plan during a selectivity learning phase [8, 72, 6, 49] or decide to use different operators to execute different portions of the data [9, 14, 77]. While these attempts have certainly proved to be effective, but they require huge modifications to the executor module and they are also limited in their ability to change join-order due to plan state management requirements.

Discussion Despite all the efforts to improve selectivity estimation module, query optimizers continue to use simplifying assumptions (AVI and value containment in joins) in difficult

estimation scenarios for the sake of efficiency of query optimization. Due to this, the process of deciding execution plan is usually associated with uncertainty wrt the selectivities and other run-time conditions. It is well known that, such inherent uncertainty present in query processing module, frequently cause its performance to be highly sub-optimal as well as unpredictable. Overall, query processing has not been *robust* against selectivity estimation errors.

2.4 Performance Metric Based Classification

In recent seminar [44] different proposals were considered regarding a metric for robustness in query processing, including: (1) coefficient of variation of execution times, (2) average relative error in cardinality estimation across all physical operators of query or (3) geometric mean of the output cardinality for different selectivities of the query. But a consensus could not be reached, possibly because robustness is dependent upon complex interaction among many factors. In this thesis, we focus only on the performance side and do not include the input parameters in the metric. Specifically, we propose a set of sub-optimality based performance metrics (details in Chapter 3) that include worst case and average case performance.

We emphasize that our goal of minimizing the worst case performance sub-optimality in the presence of unbounded selectivity errors, does not coincide with any of the earlier works in this area. Previously considered objectives in literature include: (a) improved performance compared to the optimizer generated plan [12, 47, 52, 65, 72]; (b) improved average performance and/or reduced variance [29, 21, 10]; (c) improved accuracy of selectivity estimation structures [7]; (d) bounded impact of multiplicative estimation errors [69]; and (e) smooth performance degradation [42, 16].

Further, the presence of compile-time guarantees on execution performance distinguishes the plan bouquet approach from all the previous approaches.

2.5 Approach Based Classification

While it is not completely fair to compare techniques with widely different objectives, we still provide a brief review of the previous attempts to put the plan bouquet approach in perspective. As summarized in Figure 2.1, we classify the techniques on the basis of whether they take traditional, reactive, proactive or non-traditional approach to counter the selectivity estimation errors during their compile-time and/or run-time phase. Next, we discuss the common characteristics for each of these approaches.

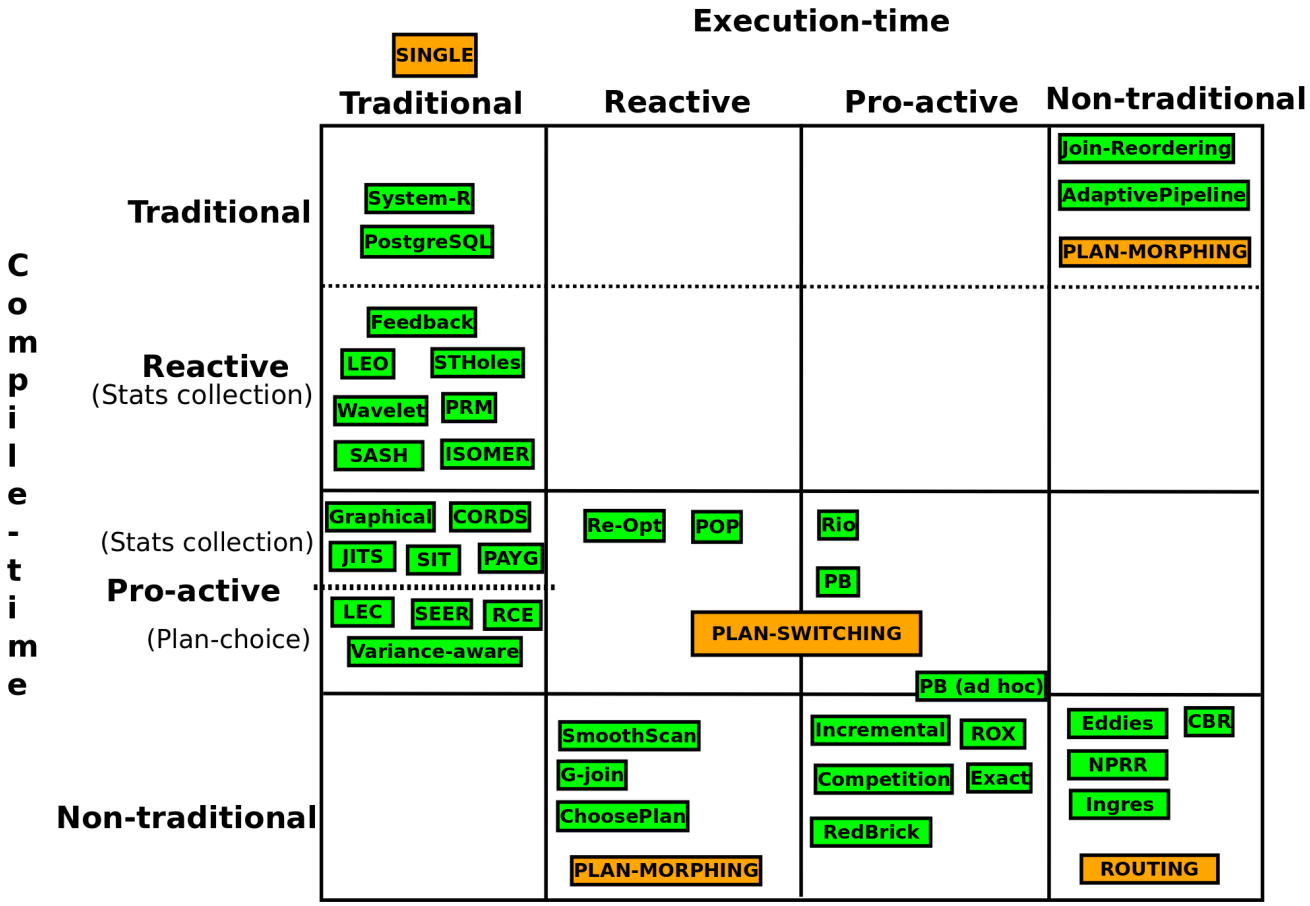


Figure 2.1: Classification on the basis of approach to handle errors

Traditional approach:

- follows basic steps of System-R approach, i.e., (a) cost-based optimization to choose an

execution plan with standard relational operators, (b) the execution steps are followed without any modification.

- performs optimally in the absence of selectivity estimation errors but can be arbitrarily sub-optimal in the face of unbounded errors.

Reactive:

- uses information available due to execution-feedback of current or previous queries.
- performs optimally in the absence of estimation errors since the only overhead is possibly a lightweight logic for learning from feedback at compile time and for error-detection at run-time.
- usually improves over the execution performance in the presence of estimation errors but is also suspectible to inconsistent selectivity inputs [80] during optimization (since selectivities are a mix of estimates and run-time feedback values) and hence prone to thrashing during execution [12].

Pro-active:

- consists of additional step(s) to prepare itself against the selectivity estimation errors and for the same reason, its total overheads are more than optimal execution even in the absence of estimation errors
- has more capability to handle estimation errors compared to reactive approaches but usually requires fresh modifications to the existing system e.g. Rio [12], Graphical models [85], etc.

Non-traditional:

- nowhere similar to System-R approach, optimizer and executor usually work together or cannot be even differentiated, since they favor adaptivity over optimality of execution.
- the performance characteristics are either similar to that of proactive approaches or not comparable since there is no optimal execution plan similar to traditional approaches, but these solution need maximum amount of modifications or even complete redesign of the

database system, e.g. Eddies [9], NPRR [73], etc.

The approach of plan bouquet technique is *proactive* during compile-time (bouquet identification step) as well as run-time (preplanned switching of plans) to counter the possibility of selectivity estimation errors. But the required implementation effort is low inspite of the proactive approach since the necessary API features have already found expression in modern database systems.

2.6 Execution-style Based Classification

The techniques can be further categorized, in terms of their execution style as (a) single plan approaches, (b) plan-switching, (c) plan-morphing and (d) tuple-routing.

Among these, all the single plan approaches, surveyed in [45], can be used in complement with the plan bouquet approach as improved estimates can reduce the number of error-prone predicates in the query. Further, the techniques that are non-traditional (in one of the phases) bring new advantages with them but also require huge implementation effort, hence they are not comparable to approaches that can be implemented in existing systems. Finally, while the plan-morphing and plan-switching techniques are already surveyed in [34, 11], we present them in comparison to the bouquet technique.

Plan-switching approaches We start with the overview of the closely related techniques which can be collectively termed as *plan-switching approaches*, as they involve run-time switching among complete query plans. At first glance, our bouquet approach, with its partial execution of multiple plans, may appear very similar to run-time re-optimization techniques such as POP [65] and Rio [12]. However, there are key differences: Firstly, they start with the optimizer's estimate as the initial seed, and then conduct a full-scale re-optimization if the estimate are found to be significantly in error. In contrast, we always start from the origin of the selectivity space, and directly choose plans from the bouquet for execution without invoking the optimizer again. A beneficial and unique side-effect of this start-from-origin approach is

that it assures repeatability of the query execution strategy.

Secondly, both POP and Rio are based on heuristics and do not provide any performance bounds. In particular, POP may get stuck with a poor plan since its validity ranges are defined using structure-equivalent plans only. Similarly, Rio's sampling-based heuristics for monitoring selectivities may not work well for join-selectivities and its definition of plan robustness on the basis of performance at corners (principal diagonal) has not been justified.

Recently, a novel interleaved optimization and execution approach was proposed in [72] wherein plan fragments are selectively executed, when recommended by an error propagation framework, to guard against the fallout of estimation errors. The error framework leverages an elegant histogram construction mechanism from [69] that minimizes the multiplicative error. While this technique substantively reduces the execution overheads, it provides no guarantees as it is largely based on heuristics.

Single plan approaches Techniques that use a single plan during the entire query execution [29, 10, 47, 69, 21] run into the basic infeasibility of a single plan to be near-optimal across the entire selectivity space. The bouquet mechanism overcomes this problem by identifying a small set of plans that collectively provide the near-optimality property. Further, it does not require any prior knowledge of the query workload or the database contents.

Our technique may superficially look similar to PQO techniques, (e.g. PPQO [15]), since a set of plans are identified before execution by exploring the selectivity space. The primary difference is that these techniques are useful for saving on optimization time for query instances with known parameters and selectivities. On the other hand, our goal is to regulate the worst case performance impact when the computed selectivities are likely to be erroneous.

Plan-morphing approaches Further, the bouquet technique does not modify plan structures at run-time (modulo spilling directives). This is a major difference from "plan-morphing" approaches, where the execution plan may be substantially modified at run-time using custom-designed operators, e.g. *chooseplan* [32], *switch* [12], *feedback* [24], etc.

One more direction that has received interest in recent years is to invent *adaptive operators* for scan [16] and joins [42] in query plans. Although, they have been shown to be quite effective in cases when the sub-optimality of the plan is a result of wrong operator decisions but not much progress has been made for cases when the sub-optimality is caused due to wrong choice of join-order itself. The primary motivation for these approaches is that the performance degradation need to be *smooth*, i.e, avoid the basic issue of *performance cliffs* in plan-switching techniques. However, in our case, the performance cliffs have been found to be quite infrequent as well as *dwarf*.

Routing-based approaches On the other hand, the use of only one active plan (at a time) to process the data makes the bouquet algorithm dissimilar from *Routing-based approaches* wherein different data segments may be routed to different simultaneously active plans – for example, plan per tuple [9] and plan per tuple group [77].

2.7 Summary

Overall, the plan bouquet technique brings performance guarantees due to its proactive approach but remains easy to deploy since the execution phase still uses traditional-plans. On the other hand, the proactive approach makes it comparatively less suitable to the environments where the estimation errors are known to be very small and the plan-switching based execution style makes it unsuitable for *latency-sensitive* applications, as discussed later in Chapter 11.

Chapter 3

Problem Framework, Notations and Assumptions

In this chapter, we present our query model, robustness model, the associated performance metrics, underlying assumptions and the notations used in the sequel.

3.1 Query Model

In our framework, each user query Q is associated with a set of selectivity predicates SP, a subset of which are error-prone wrt their estimation. Next, we define a query space QS for Q to be $\{\mathsf{Q}, AKP, EPP\}$, where AKP is the set of predicates with accurately known selectivities, and EPP is comprised of the remaining predicates that are error-prone (i.e. $AKP \cup EPP = SP$).

From the EPP, we construct an *error-prone selectivity space*, called ESS, wherein each error-prone predicate maps to an independent $[0, 1]$ selectivity dimension in the space. That is, ESS is a $[0, 1]^D$ hypercube with $D = |EPP|$, where each D-dimensional point $q(s_1, s_2, \ldots, s_D)$ represents a possible location of the query Q, as determined by its selectivities on each of these dimensions.The assignment of an independent dimension to each EPP is in conformity with the

selectivity independence assumption that is prevalent in modern query optimizer frameworks.

In the ESS defined as above, the cost of an execution plan P_i at a query location q in the ESS is denoted by $c(P_i, q)$. Also, we denote the query optimizer's *estimated* location of Q in the ESS by q_e, and the *actual* location at runtime by q_a. The optimal plan at q_e, as determined by the native optimizer, is denoted by $P_{opt}(q_e)$, and similarly the optimal plan at q_a by $P_{opt}(q_a)$. Further, we assume that the query locations and the associated estimation errors range over the *entire* ESS, that is, all (q_e,q_a) error combinations are possible.

3.2 Robustness Model

Robustness can be defined in many different ways and there is no universally accepted metric [44] – here, we use the notion of *performance sub-optimality* to characterize robustness.

With the above query model, the sub-optimality incurred due to using plan $P_{opt}(q_e)$ at location q_a is simply defined as the ratio:

$$SubOpt(q_e, q_a) = \frac{c(P_{opt}(q_e), q_a)}{c(P_{opt}(q_a), (q_a))} \quad \forall q_e, q_a \in \mathsf{ESS} \tag{3.1}$$

with $SubOpt$ ranging over $[1, \infty)$. The worst-case $SubOpt$ for a given q_a is defined to be wrt the q_e that results in the maximum sub-optimality, that is, where selectivity inaccuracies have the maximum adverse performance impact:

$$SubOpt_{worst}(q_a) = \max_{q_e \in \mathsf{ESS}} (SubOpt(q_e, q_a)) \quad \forall q_a \in \mathsf{ESS} \tag{3.2}$$

With the above, the global worst-case is simply defined as the (q_e, q_a) combination that results in the maximum value of $SubOpt$ over the entire ESS, that is:

$$\mathbf{MSO} = \max_{q_a \in \mathsf{ESS}} (SubOpt_{worst}(q_a)) \tag{3.3}$$

The above definitions are appropriate for the manner in which modern optimizers operate, wherein selectivity estimates are made at compile-time, and a single plan is executed at run-time. However, in the plan bouquet technique, neither of these characteristics is true – error-prone selectivities are not estimated at compile-time, and multiple plans may be invoked at run-time. Notwithstanding, we can still compute the corresponding statistics by: (a) substituting q_e with a "don't care" $*$; and (b) having the cost of the bouquet, denoted by $c(B, q_a)$, include the overheads incurred by the exploratory partial executions. That is,

$$SubOpt(*, q_a) = \frac{c(B, q_a)}{c(P_{opt}(q_a), q_a)} \qquad \forall q_a \in ESS \tag{3.4}$$

and

$$\mathbf{MSO} = \max_{q_a \in ESS}(SubOpt(*, q_a)) \tag{3.5}$$

Finally, the bouquet technique also furnishes a *guarantee* on its MSO performance, which is denoted by $\mathbf{MSO_g}$.

Analogous to the above, the *randomized* variants of the bouquet algorithm are evaluated for the *maximum expected sub-optimality* across ESS, defined as

$$\mathbf{MESO} = \max_{q_a \in \mathsf{ESS}}(E[SubOpt(*, q_a)])$$

and the guarantee on maximum expected sub-optimality is denoted by $\mathbf{MESO_g}$.

3.2.1 Ancillary Performance Metrics

In addition to the above primary metrics, we also evaluate the bouquet technique over a related set of performance metrics. Specifically, if we assume that all query locations and error combinations are equally likely, that is, the estimated query locations and the actual query locations are uniformly and independently distributed over the entire ESS, the *average* sub-optimality

over ESS is defined as:

$$\textbf{ASO} = \frac{\sum_{q_e \in ESS} \sum_{q_a \in ESS} SubOpt(q_e, q_a)}{\sum_{q_e \in ESS} \sum_{q_a \in ESS} 1} \tag{3.6}$$

And, the corresponding version for the bouquet technique is:

$$\textbf{ASO} = \frac{\sum_{q_a \in ESS} SubOpt(*, q_a)}{\sum_{q_a \in ESS} 1} \tag{3.7}$$

These definitions can easily be extended to the general case where the estimated and actual locations have idiosyncratic probability distributions.

An important point to note is that even when the bouquet algorithm performs well on the MSO and ASO metrics, it is possible that for some specific locations $q_a \in \mathsf{ESS}$, its performance is poorer than the worst performance of the native optimizer – that is, the bouquet is *harmful* for the queries associated with these locations. This possibility is captured using the following *MaxHarm* metric:

$$\textbf{MH} = \max_{q_a \in \mathsf{ESS}} \left(\frac{SubOpt(*, q_a)}{SubOpt_{worst}(q_a)} - 1 \right) \tag{3.8}$$

Note that MH values lie in the range $(-1, \mathsf{MSO_g} - 1]$, and harm occurs whenever MH is positive.

3.3 Notations

For notational convenience, we will hereafter represent the optimal cost and the bouquet cost for a given location q with $c_{opt}(q)$ and $c_B(q)$, respectively. Inclusive of these, the common notations used in the thesis are enumerated in Table 3.1 for quick reference.

Notation	Description
Q	User query
ESS	Error-prone Selectivity Space
D	Number of ESS dimensions
$q(s_1, s_2, ..., s_D)$	Query Location in ESS
q_e	Optimizer estimated selectivity location in ESS
q_a	ESS location corresponding to actual runtime selectivities
$P_{opt}(q)$	Optimal plan at location q
$c_{opt}(q)$	Cost of optimal plan at location q
$c_B(q)$	Cost incurred by plan bouquet for location q
$c(P_i, q)$	Cost of plan P_i at location q
$SubOpt_{worst}(q_a)$	Worst case native sub-optimality for location q_a
$SubOpt(*, q_a)$	Sub-optimality of location q_a for plan bouquet
MSO	Worst case sub-optimality across ESS
ASO	Average sub-optimality across ESS
MH	Maximum harm across ESS
$\mathsf{MSO_g}$	Compile-time guarantee on worst-case sub-optimality
$\mathsf{MESO_g}$	Compile-time guarantee on maximum expected sub-optimality
IC_k	k^{th} isosurface (isocost surface) in the ESS
$cost(IC_k)$	Cost-budget corresponding to isosurface IC_k

Table 3.1: Reference table for Notations

3.4 Assumptions

Plan Cost Functions

An assumption that fundamentally underlies the entire bouquet mechanism is that of *Plan Cost Monotonicity* (PCM) – that is, the costs of the POSP plans increase monotonically with increasing selectivity values. It captures the intuitive observation that when more data is processed by a plan, signified by larger selectivities, the cost of processing also increases. This assumption has often been made in the literature [15, 21, 46], and generally holds for the plans generated by current database systems on decision-support queries [81]. The only exception that we have found is for queries featuring *existential* operators, where the POSP plans may exhibit *decreasing* monotonicity with selectivity. Even in such scenarios, the basic bouquet technique can be utilized by the simple expedient of plotting the ESS with $(1 - s)$ instead of

s on the selectivity axes. Thus, only queries having optimal cost surfaces with a maxima or minima in the *interior* of the error space, are not amenable to our approach.

Apart from monotonicity, we also assume the cost functions to be *continuous (smooth)* throughout the ESS, again a commonplace feature in practice.

Note: We wish to highlight here that the requirement for our techniques is monotonicity and smoothness of the optimal cost profile. The existence of these properties on individual plan costs is sufficient but not necessary for the optimal cost profile to be smooth and monotonic.

In addition to the above, we also make the following assumptions during most of the thesis before relaxing them in Chapter 11.

1. **Selectivity Independence** We assume that the selectivities for the EPP's are independent of each other allowing us to construct an ESS with a different dimension for each of the EPP predicates. Later in Chapter 11, we show how the bouquet approach can be extended to provide guarantees even when the independence between predicates is not assumed.

2. **Perfect Cost Model** While the errors in cost modeling can also cause mistake in plan choices and hence performance suboptimality but since it is an orthogonal issue, for now we assume the cost model to be perfect. Intuitively, it means that the we assume a functional mapping between the abstract optimizer cost values and wall clock-times. Later in Chapter 11, we derive the impact of having an erroneus cost model on the sub-optimality based performance of the proposed technique.

3. **No known Selectivity Bounds** During this thesis, we have assumed the native engines can face arbitrarily large estimation errors and there are no lower/upper bounds available on the actual selectivities of the error-prone predicates. If these values are available in some cases, they can be used to further improve the performance of bouquet sequence as shown in Chapter 11.

Chapter 4

Robustness Bounds using Plan Bouquet Approach

We begin our presentation of the plan bouquet approach by characterizing its MSO performance bounds for the 1D scenario, and then extend the analysis to the general multi-dimensional case.

4.1 1D Selectivity Space

4.1.1 1D algorithm

By virtue of our assumptions on plan cost behavior, the PIC is a monotonically increasing and continuous function throughout the ESS; its minimum and maximum costs are denoted by C_{min} and C_{max}, respectively. As described in the Introduction, this PIC is discretized by projecting a graded progression of cost steps onto the curve. Specifically, consider the case wherein the steps are organized in a *geometric* progression with initial value a ($a > 0$) and common ratio r ($r > 1$), such that the PIC is sliced with $m = \lfloor log_r \frac{C_{max}}{C_{min}} + 1 \rfloor$ cuts, $IC_1, IC_2, \ldots IC_m$, satisfying the boundary conditions $a/r < C_{min} \leq cost(IC_1) = a$ and $cost(IC_{m-1}) < C_{max} = cost(IC_m)$, as shown in Figure 4.1.

For $1 \leq k \leq m$, denote the selectivity location where the k^{th} cost step (IC_k) intersects

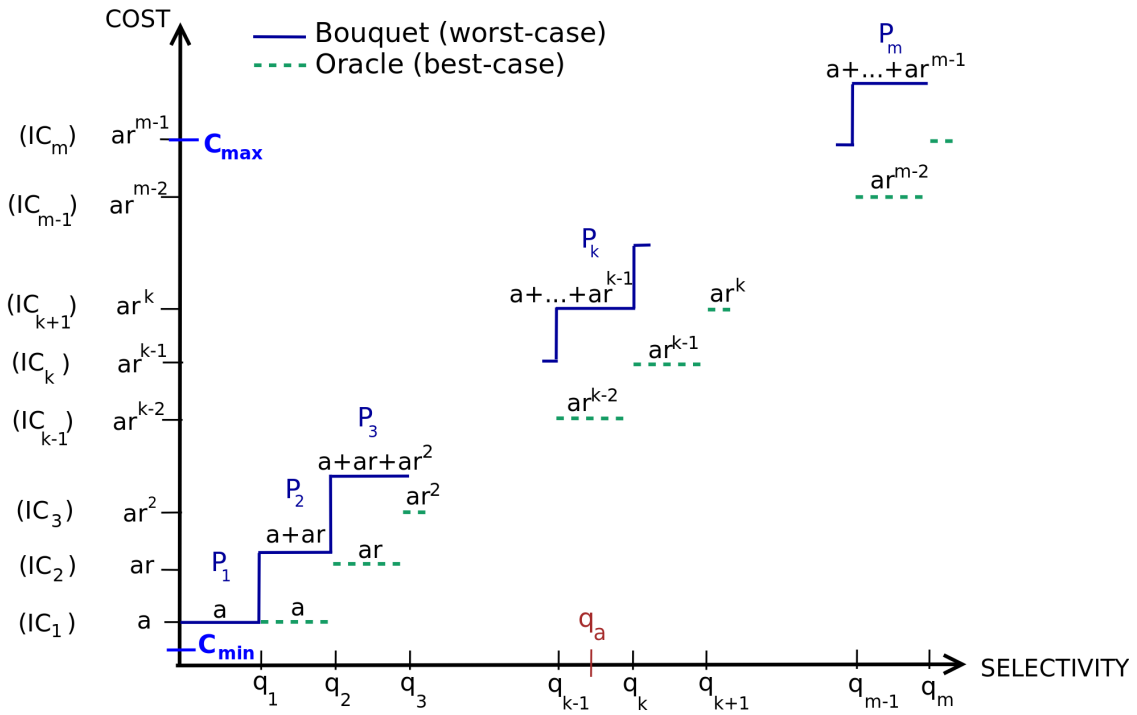


Figure 4.1: 1D selectivity space

the PIC by q_k and the corresponding bouquet plan as P_k. All the q_k locations are unique, by definition, due to the monotonicity and continuity features of the PIC. However, it is possible that some of the P_k plans may be common to multiple intersection points (e.g. in Figure 1.5, plan P1 was common to steps IC_1 through IC_4). Finally, for mathematical convenience, assign q_0 to be 0.

With this framework, the bouquet execution algorithm, outlined in Algorithm 1, operates as follows in the most general case, where a different plan is associated with each step: We start with plan P_1 and budget $cost(IC_1)$, progressively working our way up through the successive bouquet plans P_2, P_3, ... until we reach the first plan P_k that is able to fully execute the query within its assigned budget $cost(IC_k)$. It is easy to see that the following lemma holds:

Lemma 4.1 *If q_a resides in the range $(q_{k-1}, q_k]$, $1 \leq k \leq m$, then plan P_k executes it to completion in the bouquet algorithm.*

Proof: We prove by contradiction: If q_a was located in the region $(q_k, q_{k+1}]$, then P_k could

Algorithm 1: 1D Bouquet Algorithm

```
// for each cost step IC_k
for k = 1 to m do
    start executing bouquet plan P_k
    // perform cost-budgeted execution
    while run_cost(P_k) ≤ cost(IC_k) do
        execute P_k
        if P_step completes execution then
            return query result
        end
    end
    terminate P_k and discard partial results
end
```

not have completed the query due to the PCM restriction. Conversely, if q_a was located in $(q_{k-2}, q_{k-1}]$, P_{k-1} itself would have successfully executed the query to completion. With similar reasoning, we can prove the same for the remaining regions that are beyond q_{k+1} or before q_{k-2}. □

4.1.2 Performance Analysis

Consider the generic case where q_a lies in the range $(q_{k-1}, q_k]$. Based on Lemma 1, the associated worst case cost of the bouquet execution algorithm is given by the following expression:

$$c_B(q_a) = cost(IC_1) + cost(IC_k) + ... + cost(IC_k)$$

$$c_B(q_a) = a + ar + ar^2 + ... + ar^{k-1} = \frac{a(r^k - 1)}{r - 1} \tag{4.1}$$

The corresponding cost for an "oracle" algorithm that magically apriori knows the correct location of q_a is lower bounded by ar^{k-2}, due to the PCM restriction. Therefore, we have

$$SubOpt(*, q_a) \leq \frac{\frac{a(r^k-1)}{r-1}}{ar^{k-2}} = \frac{r^2}{r-1} - \frac{r^{2-k}}{r-1} < \frac{r^2}{r-1} \tag{4.2}$$

Note that the final expression is *independent* of k, and hence of the specific location of q_a. Therefore, we can state for the entire selectivity space, that:

Theorem 4.1 *Given a query* Q *with a 1D* ESS*, and the associated PIC discretized with a geometric progression having common ratio r, the bouquet execution algorithm ensures that* $\mathsf{MSO_g} = \frac{r^2}{r-1}$

Further, the choice of r can be optimized to minimize this value – the RHS reaches its minima at $r = 2$, at which the value of $\mathsf{MSO_g}$ is **4**.

4.1.3 Optimality Analysis

The following theorem shows that the proposed 1D algorithm with $r = 2$ gives the *best* performance achievable by any deterministic online algorithm – leading us to conclude that the doubling-based discretization is the ideal solution.

Theorem 4.2 *Given a universe of cost-budgeted executions of POSP plans, no deterministic online algorithm can ensure* $\mathsf{MSO_g}$ *lower than 4 in the 1D scenario.*

Proof: We prove by contradiction, assuming there exists an optimal online robust algorithm, R* with a $\mathsf{MSO_g}$ of f, $f < 4$.

The proof is divided into two parts: First, we show that R* must be a monotonically increasing sequence of plan execution costs, $[a_1, a_2, \ldots, a_m]$; and second, we demonstrate that achieving an MSO of less than 4 requires the ratio of cumulative costs for consecutive steps in the sequence to be strictly decreasing – however, this is fundamentally impossible and hence the contradiction.

(a) Assume that R* has cost sequence $[a_1, \ldots, a_i, a_j, \ldots, a_{m+1}]$ which is sorted in increasing order except for the inversion caused by $a_j < a_i$.

Now, let us define a plan execution to be *useful* if its execution covers a hitherto uncovered region of the selectivity space. With this definition, an execution of a_j after a_i is clearly useless since no fresh selectivity ground is covered by this cheaper execution. A sample instance with reference to Figure 5, is executing P_2, which covers the selectivity region $(0, q_2)$, after P_3 which covers the region $(0, q_3)$ – this does not add any value since the latter subsumes the former.

In summary, an out-of-order execution sequence cannot provide *any* improvement over an ordered sequence, which is why a_j can be safely discarded to give a completely sorted sequence $[a_1, \ldots, a_i, \ldots, a_m]$.

(b) For the sorted execution sequence R*, denote the cumulative cost at each step with $A_j = \sum_{i=1}^{j} a_i$, and the *ratio* between the cumulative costs for consecutive steps as $Y_j = \frac{A_{j+1}}{A_j}$. Note that, by definition. $A_{j+1} > A_j$.

Now, since R* has $\mathsf{MSO_g}$ of f, the sub-optimality caused by each and every step should be at most f, that is,

$$\frac{A_{j+1}}{a_j} \leq f \quad \forall j \in [1, m)$$

and therefore

$$A_{j+1} \leq f a_j \Rightarrow A_{j+1} \leq f(A_j - A_{j-1})$$

$$\Rightarrow Y_j A_j \leq f(A_j - A_{j-1})$$

After dividing both sides with A_j, we get

$$Y_j \leq f\left(1 - \frac{1}{Y_{j-1}}\right)$$

Through elementary algebra, it is known that $\forall z > 0, \left(1 - \frac{1}{z}\right) \leq \frac{z}{4}$. Therefore, we get

$$Y_j \leq \left(\frac{f}{4}\right) Y_{j-1}$$

Since $f < 4$, it implies that the sequence Y_j is strictly decreasing with multiplicative factor < 1. With repeated application of the same inequality, we obtain

$$Y_j \leq \left(\frac{f}{4}\right)^{j-1} Y_1$$

For sufficiently large j, this results in

$$Y_j < 1 \Rightarrow A_{j+1} < A_j$$

which is a contradiction to our earlier observation that $A_{j+1} > A_j$.

□

4.2 Multi-dimensional Selectivity Space

We now move on to the general case of multi-dimensional selectivity error spaces. A sample 2D scenario is shown in Figure 4.2a, wherein the isosurfaces IC_k are represented by *contours* that represent a continuous sequence of selectivity locations (in contrast to the single location in the 1D case). Further, *multiple* bouquet plans may be present on each individual contour, as shown for IC_k wherein four plans, P_1^k, P_2^k, P_3^k, P_4^k, are the optimizer's choices over disjoint (x, y) selectivity ranges on the contour. Now, to decide whether q_a lies below or beyond IC_k, in principle *every* plan on the IC_k contour has to be executed – only if none complete, do we know that the actual location definitely lies beyond the contour.

This need for exhaustive execution is highlighted in Figure 4.2b, where for the four plans lying on IC_k, the regions in the selectivity space on which each of these plans is guaranteed

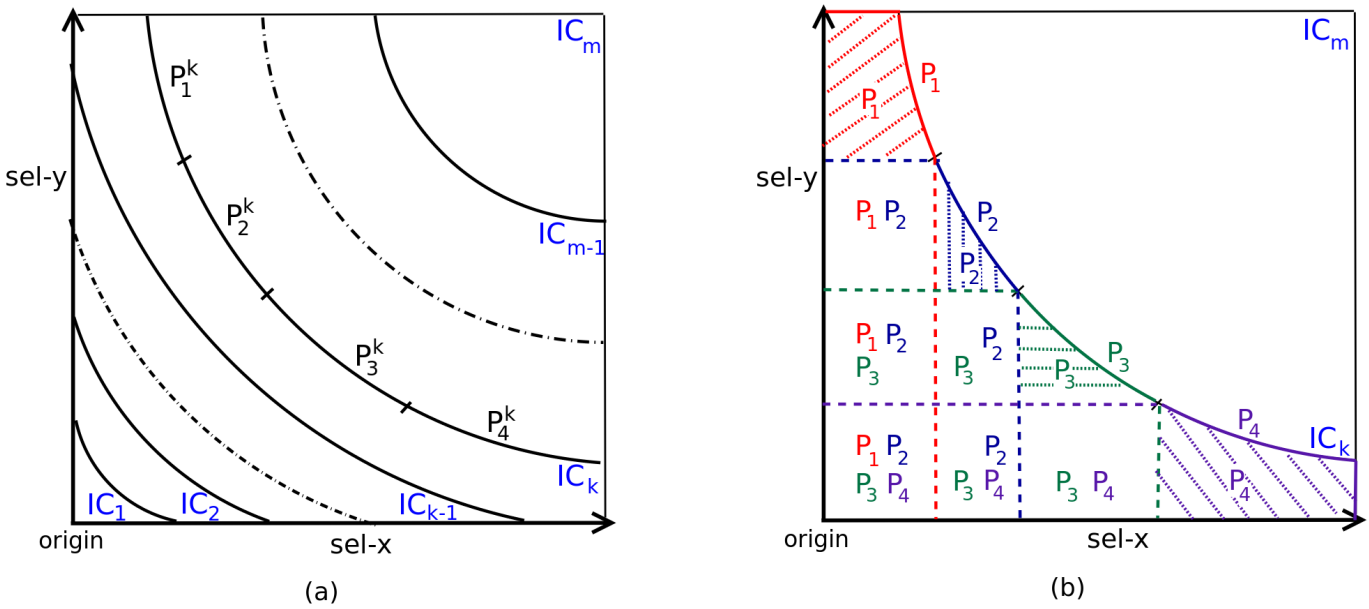


Figure 4.2: 2D Selectivity Space: (a) Isocost Contours (b) Space coverage by plans on IC_k

to complete within the budget $cost(IC_k)$ are enumerated (the contour superscripts are omitted in the figure for visual clarity). Note that while several regions are "covered" by multiple plans, each plan also has a region that it *alone* covers – the hashed regions in Figure 4.2b. For queries located in such regions, only the execution of the associated unique plan would result in confirming that the query is within the contour.

The basic bouquet algorithm for the generic multi-dimensional case is shown in Algorithm 2, using the notation n_k to represent the number of plans on isosurface IC_k.

4.2.1 Performance Bounds

Given a query Q with q_a located in the range $(IC_{k-1}, IC_k]$, the worst-case total execution cost for the multi-D bouquet algorithm is given by

$$c_B(q_a) = \sum_{i=1}^{k} [n_i \times cost(IC_i)] \tag{4.3}$$

Using ρ to denote the number of plans on the *densest* isosurface, and upper-bounding the values

Algorithm 2: Multi-dimensional Bouquet Algorithm

```
// for each isosurface IC_k
for k = 1 to m do
    // for each plan on isosurface IC_k
    for i = 1 to n_k do
        start executing bouquet plan P_i^k
        // perform cost-budgeted execution
        while run_cost(P_i^k) ≤ cost(IC_k) do
            execute P_i^k
            if P_i^k completes execution then
                return query result
            end
        end
        terminate P_i^k and discard partial results
    end
end
```

of the n_i with ρ, we get the following performance guarantee:

$$c_B(q_a) \leq \rho \times \sum_{i=1}^{k} cost(IC_i) \tag{4.4}$$

Now, following a similar derivation as for the 1D case, we arrive at the following theorem:

Theorem 4.3 *Given a query* Q *with a multidimensional* ESS*, and the associated PIC discretized with a geometric progression having common ratio r and maximum isosurface plan density ρ, the bouquet execution algorithm ensures that* $\mathsf{MSO_g} = \dfrac{\rho r^2}{r-1}$.

Setting $r = 2$ in this expression ensures that $\mathsf{MSO_g} = 4\rho$.

To the best of our knowledge, the above MSO bounds are the *first* such guarantees in the literature. Further, from these formulations, we can trivially infer that the ancillary metrics, ASO and MH, are bounded by $\mathsf{MSO_g}$ and $(\mathsf{MSO_g} - 1)$, respectively.

Chapter 5

Bounds on Maximum Expected Sub-optimality

In the previous chapter, we focused on deterministic guarantees for the worst-case sub-optimality across query locations in the entire ESS. We now move on to exploring how *randomization* can be introduced in the plan bouquet algorithm, leading to guarantees on the maximum *expected* sub-optimality, i.e., MESO_g. While our randomized algorithms work for arbitrary number of dimensions, for ease of presentation, we restrict our discussion here to 2D ESS – hence, we will use the term *contour* to represent the isosurfaces.

5.1 Randomized Intra-contour Plan Sequence

The basic plan bouquet algorithm executes n_k plans on the k^{th} contour, but does not impose any *order* on these executions. In fact, the ordering of executions has no impact on the worst case analysis, as every plan on the contour has a region that it alone covers, suggesting exactly the same worst case performance for any execution order.

Notwithstanding the above, the sub-optimality for a *particular* query instance could *vary* with the execution order of the contour plans. To analyze this, we split the bouquet overheads

into two components: (a) the overheads suffered at the finishing contour; and (b) the overheads accumulated from the earlier contours (denoted as T_B). While T_B remains the same for all query instances that lie between a consecutive pair of contours, the former is dependent on the execution order. Consider, for instance, the q_a located as shown in Figure 5.1a. With the default execution order, P_1^k through P_4^k, q_a is completed by the terminal P_4^k execution, resulting in overheads of $T_B + 4 * cost(IC_k)$. On the other hand, the overheads would reduce to $T_B + cost(IC_k)$ if P_4^k was chosen as the first plan in the execution sequence (Figure 5.1b). The implication here is that the *expected* suboptimalities can be improved by randomly choosing plan execution orders on each contour. However, minimizing this expected value may result in a weakening of the worst-case guarantee, and the tradeoff is quantified below.

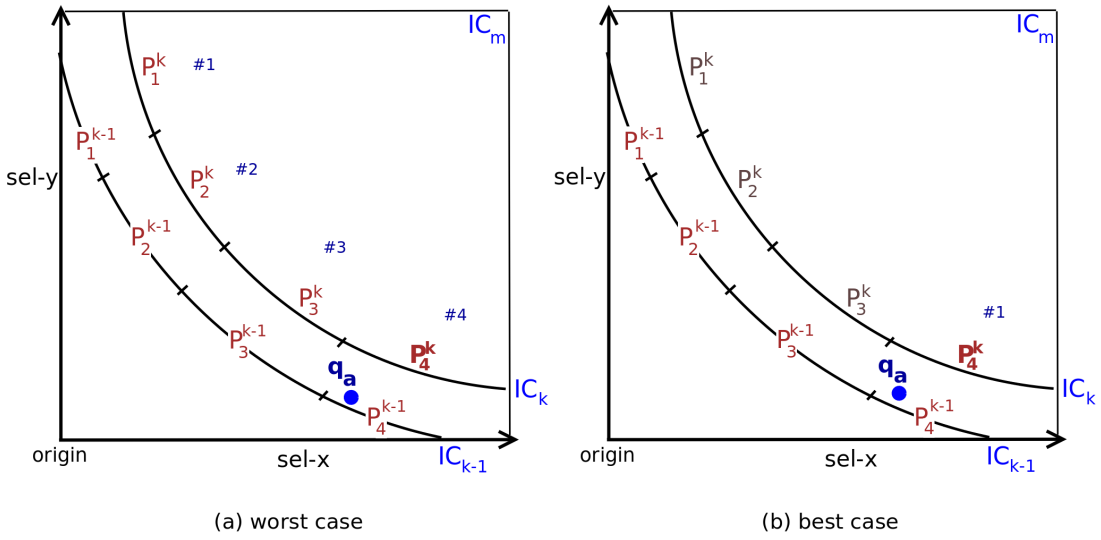


Figure 5.1: Worst-case and best-case (intra-contour) plan sequences for q_a

We construct the following variant of the bouquet algorithm – for each contour IC_k, the execution sequence of the n_k plans is a permutation chosen uniformly at random from all possible permutations. For this variant, the performance guarantees are captured by the following result:

Lemma 5.1 *The bouquet algorithm with randomized intra-contour plan sequence provides* $\mathsf{MESO_g} = \rho\left(\frac{r}{r-1} + \frac{r}{2}\right) + \frac{r}{2}$ *, while retaining* $\mathsf{MSO_g} = \frac{\rho r^2}{r-1}$.

Proof: Assume that q_a lies in the *unique* region corresponding to plan P_i^k (i^{th} plan on k^{th} contour). Note that the randomization strategy only impacts the cost incurred due to the *finishing* contour IC_k, since any permutation on the previous contours will fail to complete q_a. Specifically, since the plan sequence is chosen uniformly at random from all possible permutations, q_a will finish with $1, 2, 3, \ldots, n_k$ executions with equal probability of $\frac{1}{n_k}$. In other words, it corresponds to a discrete uniform random variable X with n_k support points, where $X = i$ represents the sequences that finish with i executions.

With the above framework, the expected bouquet cost is given by

$$E[c_B(q_a)] = cost(IC_1) + cost(IC_2) + ... + cost(IC_{k-1}) + E[cost(IC_k)]$$

leading to

$$E[c_B(q_a)] = (n_1)a + (n_2)ar + (n_3)ar^2 + ... + (n_{k-1})ar^{k-2} + \left[\frac{1}{n_k}(1 \times ar^{k-1} + ... + n_k \times ar^{k-1})\right]$$

$$E[c_B(q_a)] = (n_1)a + (n_2)ar + (n_3)ar^2 + ... + (n_{k-1})ar^{k-2} + \left[\left(\frac{n_k+1}{2}\right) \times ar^{k-1}\right]$$

Overestimating every n_i with ρ leads to

$$E[c_B(q_a)] \leq \rho(a + ar + ar^2 + ... + ar^{k-2}) + \left[\left(\frac{\rho+1}{2}\right) \times ar^{k-1}\right]$$

Dividing by ar^{k-2}, i.e., the minimum possible cost in $(IC_{k-1}, IC_k]$ gives

$$E[SubOpt(*, q_a)] \leq \rho\left(\frac{1}{r^{k-2}} + \frac{1}{r^{k-3}} + ... + r + 1\right) + \left[\left(\frac{\rho+1}{2}\right) \times r\right]$$

Finally, overestimating the finite geometric series with an infinite series provides

$$E[SubOpt(*, q_a)] < \rho\left(1 + \frac{1}{r} + \frac{1}{r^2} + ...\infty \text{ terms}\right) + \left[\left(\frac{\rho+1}{2}\right) \times r\right]$$

resulting in

$$E[SubOpt(*, q_a)] < \rho \left(\frac{r}{r-1} \right) + \left(\frac{\rho+1}{2} \right) r = \rho \left(\frac{r}{r-1} + \frac{r}{2} \right) + \frac{r}{2} \tag{5.1}$$

□

With the best possible random sequence, that requires only 1 execution for the finishing contour, the suboptimality is $\rho \left(\frac{r}{r-1}\right) + r$ and for the worst case, where ρ executions are required by the finishing contour, it is given by $\rho \left(\frac{r}{r-1}\right) + \rho r$. Clearly, the above analysis and the bound on expected sub-optimality is independent of k.

Setting a common ratio of $r = 2.4$ minimizes $\mathsf{MESO_g}$ to $2.9\rho + 1.2$ – as a side effect, $\mathsf{MSO_g}$ marginally increases from 4ρ to 4.1ρ. Moreover, even if we wish to retain $\mathsf{MSO_g}$ of 4ρ by setting $r = 2$, then $\mathsf{MESO_g}$ is only mildly weakened to $3\rho + 1$. Essentially, this suggests that we can simultaneously obtain excellent performance on both metrics.

5.2 Randomized Contour Placement

Observe that the worst case sub-optimality instances correspond to q_a's that lie *just beyond* a contour (i.e. $c_{opt}(q_a) = cost(IC_{k-1}) + \epsilon$) since their execution finishes with a plan on the next contour, which is $r-$optimal. On the other hand, q_a's that lie *just below* a contour (i.e. $c_{opt}(q_a) = cost(IC_k) - \epsilon$) complete their execution with an almost-optimal plan. Such differential treatment of query instances based on their locations can be ameliorated by randomizing the *placement* of the contours – this is illustrated in Figure 5.2 for the example location q_a. With the original contour placement (Figure 5.2a), P_4^k completes execution for q_a expending $cost(IC_k)$, whereas after slightly repositioning the contours (Figure 5.2b), q_a is completed by $P_4^{k'}$ with $cost(IC_{k'}) \approx \frac{cost(IC_k)}{r}$.

To leverage the above, we construct a randomized variant, similar to that proposed in [63], wherein the entire geometric sequence is shifted left by a random multiplicative factor $\frac{1}{r^X}$, where X is a uniform random variable $\in [0, 1)$. That is, the cost associated with the first

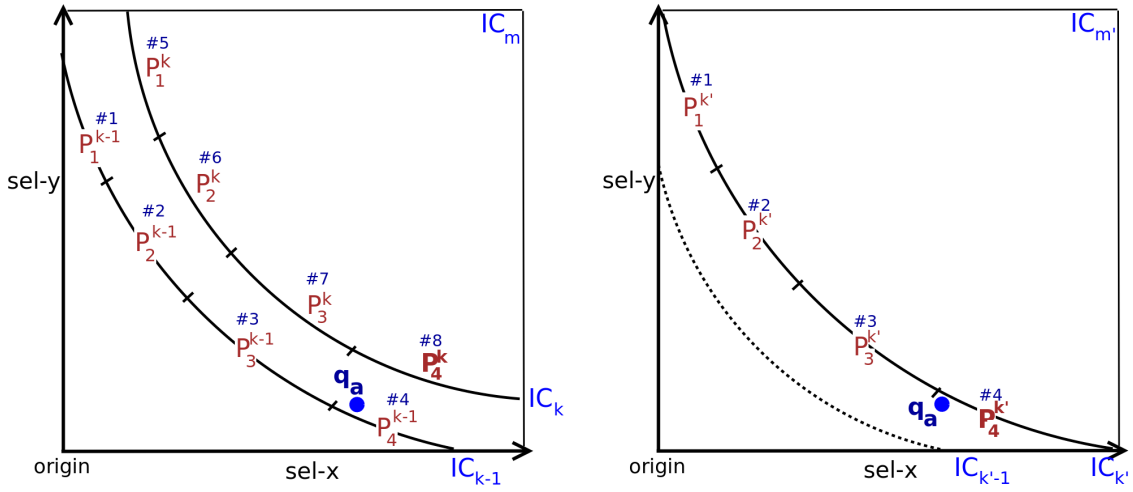


Figure 5.2: Worst-case and best-case contour placements for q_a

contour is randomized between $\frac{a}{r}$ and a, the later contours retaining the default r cost-ratio. For this variant, the performance guarantees are captured by the following result:

Lemma 5.2 *The bouquet algorithm with randomized contour placement provides* $\mathsf{MESO}_g = \rho\frac{r}{\ln r}$ *, while retaining* $\mathsf{MSO}_g = \frac{\rho r^2}{r-1}$.

Proof: For starters, assume that the contours are placed as per the deterministic bouquet algorithm. Now, consider a q_a that lies infinitesimally above IC_{k-1} – its worst-case suboptimality is $\frac{\rho r^2}{r-1}$, as per Theorem 4.3. However, when the entire geometric sequence is shifted left by a random multiplicative factor $\frac{1}{r^X}$, where X is a uniform random variable $\in [0, 1)$, this suboptimality becomes a decreasing function of the amount of shift. Specifically, the expected sub-optimality is

$$E[SubOpt(*, q_a)] < E\left[\frac{1}{r^X}\left(\frac{\rho r^2}{r-1}\right)\right] = \frac{\rho r^2}{r-1}\left(\int_{X=0}^{1} r^{-X} dX\right) = \frac{\rho r^2}{r-1} \times \left(\frac{r-1}{r \ln r}\right) = \frac{\rho r}{\ln r} \tag{5.2}$$

Caveat: An implicit assumption in the above analysis is that the value of ρ is not increased by

the randomization (although the individual contour plan densities may have been altered due to the changed contour locations).

Next we claim that, a similar analysis holds for an arbitrary q_a lying between the original deterministic contours IC_{k-1} and IC_k. The only difference is that instead of a continuously decreasing sub-optimality function of the shift, we get the following behavior: The sub-optimality initially decreases as the IC_k contour moves from its original position towards q_a, reaching a minimum when q_a is present on the contour. Then, there is a sudden discontinuous increase in sub-optimality when the contour crosses q_a, because q_a is now covered by a new contour that is r times its optimal cost. As the shift continues, the new covering contour begins to move closer to q_a, again resulting in a decreasing function. In expectation, this behavior is the same as that shown for the specific case above.

To put the above arguments mathematically, consider a generic location with optimal cost $cost(IC_{k-1}) \times r^z$ with z $\in (0, 1]$. Here, the contour IC_k comes closer to q_a when X varies in the range $[0, 1-z)$ and suddenly covered by a r-optimal contour just after $X = 1-z$, beyond which the new contour moves closer to q_a. With this behavior, the expected sub-optimality for this location is given by:

$$E[SubOpt(*, q_a)] < \int_{X=0}^{1-z} \left(\frac{1}{r^z} \times \frac{1}{r^X}\right) \left(\frac{\rho r^2}{r-1}\right) dX + \int_{X=1-z}^{1} \left(r \times \frac{1}{r^z} \times \frac{1}{r^X}\right) \left(\frac{\rho r^2}{r-1}\right) dX$$

$$E[SubOpt(*, q_a)] < \left(\frac{\rho r^2}{r-1}\right) \left[\int_{X=0}^{1-z} \left(\frac{1}{r^{X+z}}\right) dX + \int_{X=1-z}^{1} \left(\frac{1}{r^{X+z-1}}\right) dX\right]$$

Substituting $Y_1 = X + z$ and $Y_2 = X + z - 1$,

$$E[SubOpt(*, q_a)] < \left(\frac{\rho r^2}{r-1}\right) \left[\int_{Y_1=z}^{1} \left(\frac{1}{r^{Y_1}}\right) dY_1 + \int_{Y_2=0}^{z} \left(\frac{1}{r^{Y_2}}\right) dY_2\right]$$

$$E[SubOpt(*, q_a)] < \left(\frac{\rho r^2}{r-1}\right)\left[\left(\frac{-1}{\ln r}\right)\left(\frac{1}{r}-\frac{1}{r^z}\right)+\left(\frac{-1}{\ln r}\right)\left(\frac{1}{r^z}-\frac{1}{r^0}\right)\right]$$

$$E[SubOpt(*, q_a)] < \left(\frac{\rho r^2}{r-1}\right)\left[\left(\frac{1}{\ln r}\right)\left(1-\frac{1}{r}\right)\right] = \frac{\rho r}{\ln r}$$

□

Setting a common ratio of $r = e \approx 2.72$ minimizes MESO_g to $\approx 2.72\rho$ – as a side effect, MSO_g slightly increases to 4.3ρ. Moreover, even if we wish to retain the 4ρ MSO guarantee by setting $r = 2$, then MESO_g is only mildly weakened to 2.89ρ. Again, we observe simultaneous excellent performance on both metrics.

5.3 Using the Randomization Strategies in Tandem

We now move on to deriving MESO_g when *both* randomization strategies are applied to the plan bouquet algorithm in tandem. Specifically, the contour placement randomization is applied first, and then for each resulting contour, the execution order randomization is applied. For this combined algorithm, the following theorem gives an upper bound on the worst-case expected sub-optimality.

Theorem 5.1 *Given a query* Q *on a multi-dimensional* ESS*, the bouquet execution algorithm with contour placement randomization followed by intra-contour plan sequence randomization provides* $\mathsf{MESO}_g = \rho\frac{(r+1)}{2\ln r} + \frac{(r-1)}{2\ln r}$ *, while retaining* $\mathsf{MSO}_g = \frac{\rho r^2}{r-1}$ *.*

Proof: The intra-contour randomization performance for a given placement of contours (Lemma 5.1), remains essentially the same for each instantiated value of the contour placement random variable (Lemma 5.2). Hence, we use the *law of iterated expectations* to calculate the expected sub-optimality for a given location q_a, by leveraging Equation 5.1 across different random contour placements:

$$E[SubOpt(*, q_a)] \leq E\left[\frac{1}{r^X}\left[\rho\left(\frac{r}{r-1}+\frac{r}{2}\right)+\frac{r}{2}\right]\right]$$

$$E[SubOpt(*, q_a)] \leq \left[\rho\left(\frac{r}{r-1}+\frac{r}{2}\right)+\frac{r}{2}\right] \times E\left[\frac{1}{r^X}\right]$$

$$E[SubOpt(*, q_a)] \leq \left[\rho\left(\frac{r}{r-1}+\frac{r}{2}\right)+\frac{r}{2}\right] \times \frac{r-1}{r\ln r}$$

$$E[SubOpt(*, q_a)] \leq \left[\frac{r(r+1)\rho}{2(r-1)}+\frac{r}{2}\right] \times \frac{r-1}{r\ln r}$$

$$E[SubOpt(*, q_a)] \leq \frac{r+1}{2\ln r}\rho + \frac{r-1}{2\ln r} = \frac{(r+1)\rho+(r-1)}{2\ln r} \tag{5.3}$$

□

For $r = 2$, we obtain

$$|3\rho+1|_{r=2} \times 0.72 = |2.16\rho+0.72|_{r=2}$$

Further, this guarantee bound can be improved by minimizing the $\frac{r+1}{\ln r}$ multiplier of ρ – the multiplier reaches its minimum value for $r = 3.6$, leading to

$$\mathsf{MESO_g} = |1.8\rho+1|_{r=3.6}$$

5.4 Discussion

The above results are summarized in Table 5.1. It is noteworthy that using $r = 2$ retains $\mathsf{MSO_g}$ while minimizing $\mathsf{MESO_g}$ requires different cost-ratios for each variant.

With regard to implementation, the intra-contour plan sequence only requires a simple shuffling algorithm – for instance, the standard *Knuth's shuffle* [54]. On the other hand, random-

Variant	Cost-ratio(r)	MESO_g	MSO_g
No randomization	2	4ρ	4ρ
Randomized Plan Sequence	2	$3\rho + 1$	4ρ
	2.4	$2.9\rho + 1.2$	4.1ρ
Randomized Contour Placement	2	2.89ρ	4ρ
	2.4	2.74ρ	4.1ρ
	2.72	2.72ρ	4.3ρ
Randomized Plan Sequence & Contour Placement	2	$2.16\rho + 0.72$	4ρ
	2.4	$1.94\rho + 0.8$	4.1ρ
	2.72	$1.86\rho + 0.86$	4.3ρ
	3.6	$1.8\rho + 1$	4.98ρ

Table 5.1: Performance of randomized variants of the bouquet algorithm

izing the initial contour location is more complicated since in principle, for each new starting cost, a complete rescan of the ESS is required to determine the fresh set of contours. However, even if we restricted ourselves to merely *two* instances of contour placement, corresponding to $X = 0$ and $X = 0.5$, we achieve an attractive combination of $\mathsf{MESO}_g = 2.38\rho + 1$ and $\mathsf{MSO}_g = 4.1\rho$, for $r = 2.4$.

Chapter 6

Compile-Time Enhancements to Improve Robustness Bounds

The bouquet mechanism's MSO_g guarantee of 4 for the 1D case is shown to be inherently strong in Section 4.1.1. However, the multi-dimensional bounds depend on ρ, the maximum plan density across the isosurfaces, which can be quite high – for instance, in excess of 150 for the 5D queries considered in our study. Therefore, to have a practically useful bound, we need to ensure that the value of ρ is reduced as far as possible.

A potential approach to achieving reduction in the "effective" value of ρ is to somehow skip some of the cost-budgeted executions from the original bouquet sequence. At first glance, such removal of executions may appear contrary to the principle of *exhaustive* contour execution described in Section 4.2. However, as we will show in the remainder of this section, it can be achieved by ensuring that the roles of skipped executions are played by carefully identified alternative executions. Specifically, we present two compile-time enhancements here for implementing such an execution skipping process.

6.1 Plan Swallowing Enhancement

Our first technique leverages the notion of "anorexic reduction" [46] to directly reduce the cardinality of the POSP itself. In this approach, POSP plans are allowed to "swallow" other plans, that is, occupy their regions in the ESS, if the sub-optimality introduced due to these swallowings can be bounded to a user-defined threshold, λ. Through extensive experimentation, it was shown in [46] that even for complex OLAP queries with high dimensional ESS, a λ setting of 20% was typically sufficient to bring the number of POSP plans down to "anorexic levels", that is, a small absolute number within or around 10.

When anorexic reduction is introduced into the plan bouquet setup, it immediately serves to steeply reduce the effective value of ρ. However, there is also a downside – the constant multiplication factor is increased by a factor $(1 + \lambda)$ due to the inflation in the cost budget. Overall, the deterministic guarantee is altered from $4\,\rho_{\text{POSP}}$ to $4\,(1+\lambda)\,\rho_{\text{ANOREXIC}}$.

Empirical evidence that this tradeoff is highly beneficial is shown in Table 6.1, which compares for a variety of multi-dimensional error spaces, the bounds (using Equation 4.3) under the original configuration and under anorexic reduction ($\lambda = 20\%$). As a particularly compelling example, consider 5D_DS_Q19, a five-dimensional selectivity error space based on Q19 of TPC-DS – we observe here that MSO_g plunges by more than an order of magnitude, going down from 379 to 30.4.

6.2 Execution Covering Enhancement

We now move on to describing an independent and complementary enhancement that can further reduce the effective ρ. It leverages the observation that even if a particular execution is skipped, the selectivity region covered by this execution can still be covered using execution(s) from *later* contours – of course, at a higher cost. Such skipping clearly implies increase in sub-optimality for some individual query instances – however, from a holistic perspective, it serves

Error Space	ρ_{posp}	MSO_g	ρ_{anorexic}	MSO_g
3D_H_Q5	11	**33**	3	**12.0**
3D_H_Q7	13	**34**	3	**9.6**
4D_H_Q8	88	**213**	7	**24.0**
5D_H_Q7	111	**342.5**	9	**37.2**
3D_DS_Q15	7	**23.5**	3	**12.0**
3D_DS_Q96	6	**22.5**	3	**13.0**
4D_DS_Q7	29	**83**	4	**17.8**
4D_DS_Q26	25	**76**	5	**19.8**
4D_DS_Q91	94	**240**	9	**35.3**
5D_DS_Q19	159	**379**	8	**30.4**

Table 6.1: Effect of Anorexic Reduction [$\lambda = 20\%$] on Robustness Guarantees

to substantively reduce the effective ρ and thereby deliver much stronger MSO_g guarantees.

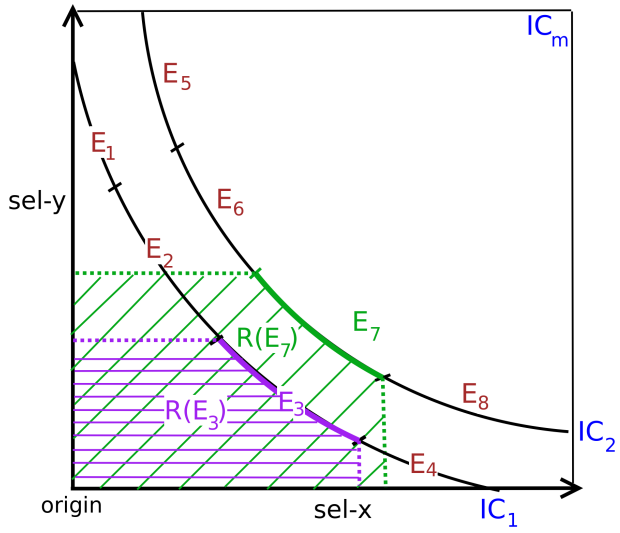


Figure 6.1: E_7 can complete all locations in $R(E_3)$ with cost(IC_2) = 2 * cost(IC_1)

To formalize this enhancement, we represent the bouquet algorithm as a sequence of cost-budgeted plan executions $BS = \{E_1, E_2, \ldots, E_{terminal}\}$, where $E_{terminal}$ is the final execution that can complete all locations in the ESS. Additionally, we use the function $\phi(E_i)$ to indicate the identity of the executed plan, and $\omega(E_i)$ to represent the cost budget of the execution. So, if E_i corresponds to the execution of P_j^k, then $\phi(E_i) = P_j$ and $\omega(E_i) = cost(IC_k)$.

Now, for each E_i, denote with $R(E_i)$, the *region* of the ESS that E_i is apriori known to

certainly complete within its budget, i.e, all q s.t. $c(\phi(E_i), q) \leq \omega(E_i)$. To make this notion concrete, visual representations of $R(E_3)$ and $R(E_7)$ are shown in Figure 6.1, highlighted with purple horizontal lines and green slanted lines, respectively. In addition, the figure also shows that E_7 can complete all query locations in $R(E_3)$ within twice the cost budget of E_3.

For quick reference, the notations employed hereafter in this section are summarized in Table 6.2.

Notation	Description
$\succeq_{cover}$	Execution 'cover' relation
BS	Original bouquet execution Sequence
E_i	i^{th} execution in the bouquet execution sequence
$E_{terminal}$	Final execution in the execution sequence
$\phi(E_i)$	Identity of plan used in execution E_i
$\omega(E_i)$	Cost budget of execution E_i
$R(E_i)$	Region in ESS covered by i^{th} execution
$SubOpt(E_i)$	SubOpt corresponding to i^{th} execution
CS	Covering execution Sequence
CS^k	Covering Sequence that covers the set of original executions from IC_k
CS_{opt}	Set of Optimal Covering Sequence(s)
CSI	Covering Sequence Identification Algorithm

Table 6.2: Reference table for Notations for Execution Covering

6.2.1 The Cover Relation

We define the ability of an execution to complete the ESS region of another execution as the *Cover Relation* over the set of executions. Formally, an execution E_i can *cover* execution E_j, denoted as $E_i \succeq_{cover} E_j$, if $R(E_i) \supseteq R(E_j)$. The $\succeq_{cover}$ relation imposes a partial order on the set of executions, with $E_{terminal}$ being the unique top element since $R(E_{terminal}) = \text{ESS}$.

Example An example 2D ESS is shown in Figure 6.2, featuring a total of 32 executions spanning across 7 contours. The corresponding *Hasse diagram* for the cover relation on the set of executions in the bouquet sequence is shown in Figure 6.3a, where elements of the partial order become nodes, and non-transitive relations among the elements become edges. Further, the weight of each node is given by the cost budget of the corresponding execution, i.e., $\omega(E_i)$.

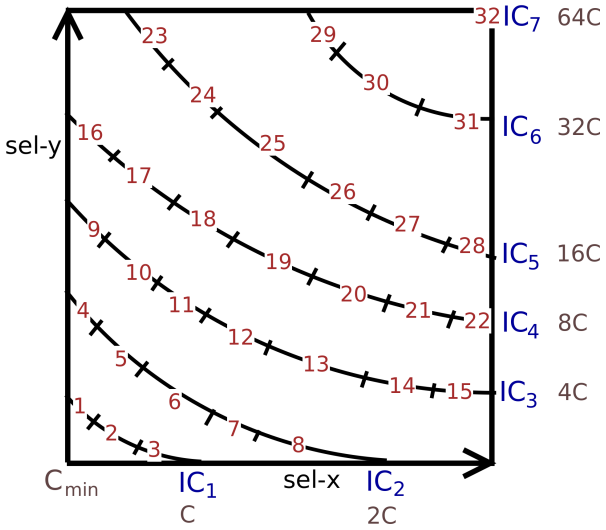


Figure 6.2: Example bouquet sequence

Since the weights are the same for all executions from a given contour, they are highlighted only once for each contour in Figure 6.3a, e.g., 8C for nodes 16 to 22.

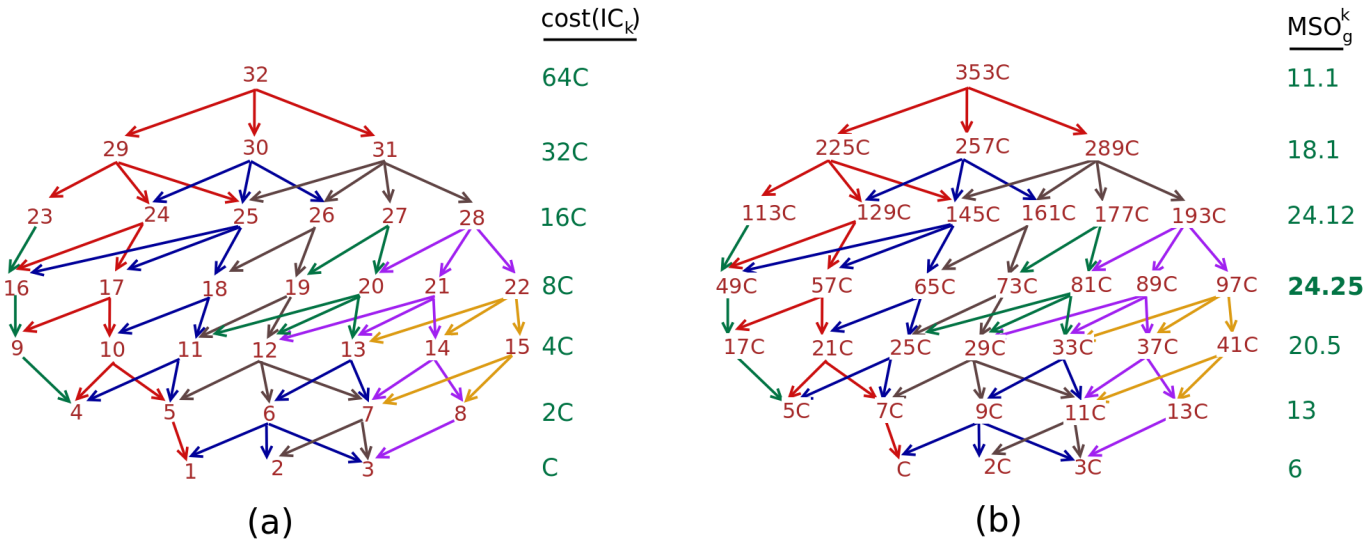


Figure 6.3: (a) Hasse diagram (b) Execution cost and sub-optimality analysis

Detailed sub-optimality analysis for the basic bouquet sequence Next, we show in Figure 6.3b, the detailed analysis of the basic bouquet execution sequence wrt execution cost and sub-optimality. Here, each node E_i is labeled with the accumulated overheads until and

including E_i, i.e. $\sum_{j=1}^{i} \omega(E_j)$. While the accumulated overheads are monotonically increasing, the sub-optimality variation is not necessarily so, and is given by the following recurrence formula:

$$SubOpt(E_i) = \begin{cases} SubOpt(E_{i-1}) + r & \text{if } \omega(E_i) = \omega(E_{i-1}) \\ \dfrac{SubOpt(E_{i-1})}{r} + r & \text{if } \omega(E_i) \neq \omega(E_{i-1}) \\ r & \text{if } i = 1 \end{cases}$$

With each new execution, the sub-optimality gets an additive term r due to the increase in the accumulated execution cost. But whenever there is a contour change, the sub-optimality first undergoes an improvement by a factor of r due to increase in the minimum optimal cost covered by E_i, before facing the additive term r. Thus, the sub-optimality increases while working our way through a contour, but may observe a dip when a contour jump happens.

Similarly, with MSO_g^k defined as the maximum sub-optimality encountered in the region between contours IC_{k-1} and IC_k, we find that MSO_g^k varies according to the recurrence: $MSO_g^k = rn_k + \dfrac{MSO_g^{k-1}}{r}$ with $MSO_g^0 = 0$ for mathematical convenience. It is noteworthy that, MSO_g^k increases monotonically with contour index k only if $n_k > \frac{MSO_g^{k-1}}{r^2}$ is satisfied for all $k \in (1, m]$. Finally, in Figure 6.3b, these computed values are marked besides each contour and the overall $\mathsf{MSO_g} = 24.25$ that occurs on the 4th contour, is highlighted in boldface.

6.2.2 Motivating Scenario: $\mathsf{MSO_g}$ Reduction due to Execution Covering

Consider Figure 6.3a, where execution E_{28} is capable of covering executions E_{20}, E_{21} and E_{22}. That is, if E_{20}, E_{21}, E_{22} are skipped from the bouquet sequence, their associated regions $\cup_{i=20}^{22} R(E_i)$ can still be covered by execution E_{28}. Implementing this observation, as depicted in Figure 6.4, the effective plan density of IC_4 reduces from 7 to 6 – since the cost budget of E_{28} is equivalent to 2 executions from IC_4. Note that, this execution cover has *no impact* on sub-optimality performance till E_{19}, but causes a sub-optimality reduction for all the later

executions and hence reduction in MSO_g^k for all the contours beyond IC_4. Overall, $\mathsf{MSO_g}$ marginally reduces from 24.25 to 22.25.

As a general rule, implementing a particular execution covering does not harm $\mathsf{MSO_g}$ if the cover's budget does not exceed the sum of the budgets of the replaced executions. That is, $E_{cover} \succeq_{cover} \{E_c, \ldots, E_{c'}\}$ can be employed if $\omega(E_{cover}) \leq \omega(E_c) + \ldots + \omega(E_{c'})$.

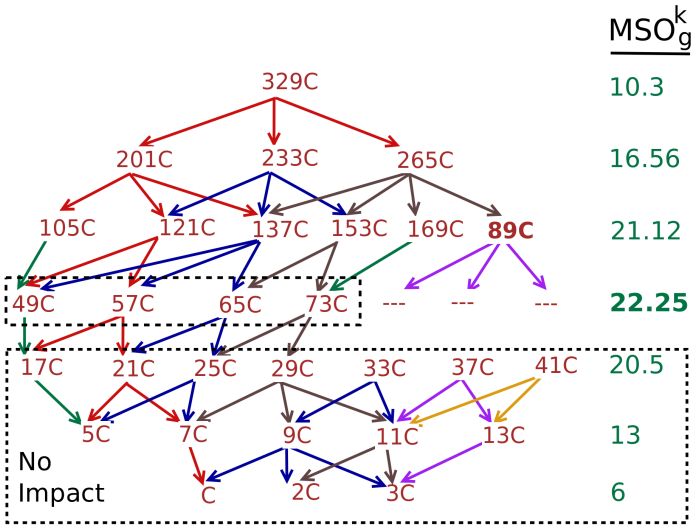


Figure 6.4: Using cover $E_{28} \rightarrow \{E_{20}, E_{21}, E_{22}\}$ improves $\mathsf{MSO_g}$ from 24.25 to 22.25

6.2.3 Covering Sequences

Extending the above single execution cover example, we define a *covering sequence* (CS) as an execution sequence that contains a cover for each and every execution in the original sequence. Since $E_{terminal}$ cannot be covered by any other execution, it must be present in every candidate covering sequence, implying a total of $2^{|BS|-1}$ candidates. For each candidate CS, $\mathsf{MSO_g}$ can be computed as

$$\mathsf{MSO_g}(CS) = \max_{1 \leq a \leq m} \left[\frac{\sum_{k=1}^{a} \Omega(CS^k)}{cost(IC_{a-1})} \right]$$

where $CS^k \subset CS$ is the set of execution(s) that cover executions from IC_k and $\Omega(CS^k) = \sum_{E_i \in CS^k} \omega(E_i)$ and m is the total number of contours. [1] Also, $cost(IC_0) = C_{min} = \frac{cost(IC_1)}{2} + \epsilon$.

Optimal Covering Sequence Among all the CS candidates, the covering sequence(s) corresponding to the minimum value of $\mathsf{MSO_g}$ are characterized as *optimal* covering sequences (CS_{opt}). The CS_{opt} sequences can be identified through a brute force evaluation of all candidates, but the complexity is exponential in the number of executions in the sequence, and it is therefore impractical. As a viable alternative, we propose a greedy algorithm, termed as *Covering Sequence Identification* (CSI), to find a CS with improved $\mathsf{MSO_g}$. Specifically, CSI decomposes the original problem into contour-wise subproblems, each of which is modeled as a *red-blue domination* problem. The subproblems are then solved efficiently using a greedy approach, and the contour-wise solution nodes are stitched together to form a covering sequence (details in Chapter 7).

As a concrete outcome of the CSI algorithm, the solution CS for the running example is shown in Figure 6.5a (the equivalent ESS coverage representation is shown in Figure 6.5b). Note that the resulting $\mathsf{MSO_g}$ has come down to only **14.5** as compared to 24.25 of the original bouquet sequence.

With regard to the above CS solution, a few interesting sidelights emerge:

- In contrast to the original bouquet sequence, the executions are not necessarily contour-ordered in a covering sequence. For instance, E_{31} from IC_6 is executed before executions E_{23} and E_{24} from IC_5.
- The CS uses only 11 out of 32 executions in the original sequence.
- The effective contour plan densities decrease from $[3, 5, 7, 7, 6, 3, 1]$ to $[2, 4, 4, 4, 2, 2, 1]$. Specifically for IC_3, ES uses 7 executions from IC_3, while CS covers it employing CS^3

[1] If an execution is capable of acting as a cover for executions from different contours, then it is counted only once for the lowest index contour, while calculating $\mathsf{MSO_g}$.

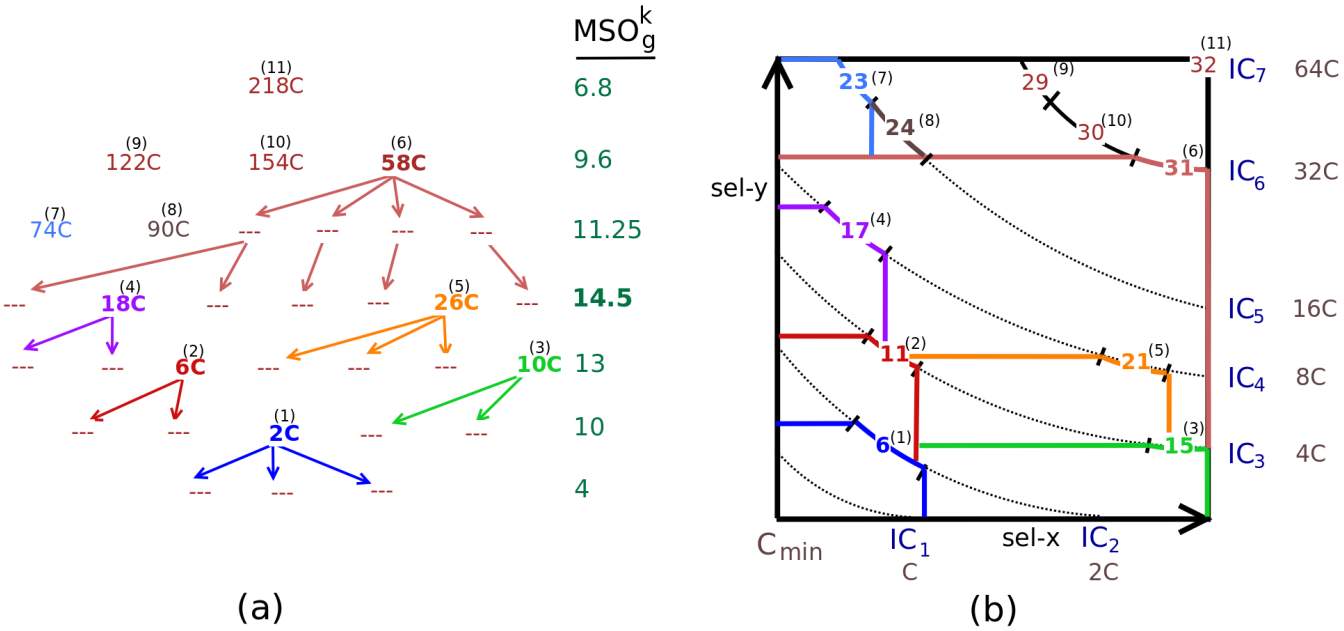


Figure 6.5: (a) Execution covering sequence (b) Modified space coverage

$=\{\underline{E_{11}}, \underline{E_{15}}, E_{17}, E_{21}\}$, which is equivalent to 4 executions (since it requires two executions, E_{17} and E_{21}, from IC_4. [1]

- An increase in execution cost (and hence sub-optimality) occurs only for the three regions covered by $E_1(C \rightarrow 2C), E_4(5C \rightarrow 6C)$ and $E_9(17C \rightarrow 18C)$. For most of the remaining regions, the execution cost improves *significantly*, e.g., $E_{15}(41C \rightarrow 10C)$ and $E_{31}(265C \rightarrow 58C)$. This observation suggests that along with significant reductions in $\mathsf{MSO_g}$, concurrent improvements in ASO and MH may also be expected.

The effectiveness of CSI is further corroborated by its performance on queries based on the TPC-H and TPC-DS benchmarks – these results are summarized in Table 6.2.3. Overall, the $\mathsf{MSO_g}$ never exceeded **20** even for high-dimensional queries, including those with high initial values of $\mathsf{MSO_g}$! As a particularly compelling example, for a five-dimensional error-space 5D_H_Q7, the covering sequence used only 10 executions (out of the original 34), and more importantly brought $\mathsf{MSO_g}$ down from 37.2 to *only 15*.

[1] The underlined executions, E_{11} and E_{15}, are part of CS^3 but their overheads are already counted in CS^2.

Error Space	MSO_g (Anorexic)	# Executions (Anorexic)	MSO_g (Anorexic+CSI)	# Executions (Anorexic+CSI)
3D_H_Q5	**12**	8	**8.4**	4
3D_H_Q7	**9.6**	6	**7.2**	3
4D_H_Q8	**24**	18	**15**	7
5D_H_Q7	**37.2**	34	**15**	10
3D_DS_Q15	**12**	16	**9.2**	9
3D_DS_Q96	**13**	10	**8.8**	7
4D_DS_Q7	**17.8**	14	**9.1**	7
4D_DS_Q26	**19.8**	23	**8.1**	8
4D_DS_Q91	**35.3**	34	**16**	14
5D_DS_Q19	**30.4**	24	**15**	13

Table 6.3: Effect of Execution Covering on Robustness Guarantees

Computational Effort Overall, the worst case complexity for the CSI algorithm is $O(m\rho \log(m\rho))$ against $O(2^{m\rho})$ of the brute force algorithm. Further, to be able to use this enhancement, it is required to first construct the Hasse Diagram which involves establishing the *cover* relation among all pairs of executions from consecutive contours. For this purpose, we have devised a three step mechanism, where the proposed checks in the first two steps are computationally very cheap as compared to the third step. To elaborate, we first identify true positives by evaluating a simple necessary and sufficient criteria, then discard true negatives by evaluating another cheap to evaluate necessary condition, and finally, if the previous two steps do not prove conclusive, evaluate the computationally expensive sufficiency criteria. The complete details of this procedure can be found in Chapter 7.

Finally, since the computational efforts required in both Hasse diagram construction and covering sequence identification, depend on the number of executions in the sequence, it is recommended to use CSI only after anorexic reduction has already been employed.

Chapter 7

Efficient Identification of Plan Bouquet Sequence

Given a user query Q, the first step is to identify the error-prone selectivity dimensions in the query. For this, we can leverage the approach proposed in [52], wherein a set of uncertainty modeling rules are outlined to classify selectivity errors into categories ranging from "no uncertainty" to "very high uncertainty". Alternatively, a log could be maintained of the errors encountered by similar queries in the workload history. Finally, there is always the fallback option of making *all* predicates where selectivities are evaluated, to be selectivity dimensions for the query.

The chosen dimensions form the error-prone selectivity space(ESS). In general, each dimension ranges over the entire [0,1] selectivity range – however, due to schematic constraints, the range may be reduced. For instance, the maximum legal value for a PK-FK join is the reciprocal of the PK relation's row cardinality. The next step is to identify the isosurfaces in this ESS.

7.1 NEXUS: Algorithm for Identifying an Isosurface

The primary inputs to the bouquet identification phase are the isosurfaces (contours in 2D) drawn on the ESS. Thus far, we had viewed each isosurface as a continuous region comprised of selectivity locations having identical cost values for their optimal plans. As a practical matter, however, we have to construct and process approximate *discretized* versions of these regions. That is, we need to use a D-dimensional grid with finite resolution[1] res to approximate the ESS hypercube $[0, 1]^D$.

With this discretized ESS, an isosurface for cost C is constructed as a D-dimensional set of contiguous grid locations q such that $c_{opt}(q)$ lies in the interval $[C, (1 + \alpha)C]$, where $c_{opt}(q)$ is the cost of the optimal plan at location q, and α is a tolerance factor. Since the tolerance factor could occasionally result in "thickening the surface" due to inter-surface locations also creeping into the surface set, we additionally require that each point in the surface must have at least one of its lower neighbors violating the above cost interval requirement. Finally, we assume that the resolution of the ESS grid is sufficiently high such that we can always find contiguous isocost locations even with small values of α, say 0.05.

A straightforward strategy to identify the isosurfaces from the ESS is to first explore the discretized ESS in a exhaustive manner, and then identify the locations that are acceptable for the required isocost values. But the overheads for such an approach would increase exponentially with ESS dimensionality, and become impractical for typical OLAP queries. Moreover, the exhaustive enumeration is an overkill for isosurface identification since: (a) we do not need information about the internal regions that lie *between* the isosurfaces and take up the vast majority of the space in ESS; and (b) we do not exploit the potential for overlapping the *identification* of later isosurfaces with the *execution* of the earlier isosurfaces, which could provide a head-start in the bouquet execution process.

[1] We use the term *resolution* to remain consistent with prior works [46, 47] that dealt with discretized selectivity spaces.

Motivated by the above observations, we propose[1] in this section a *focused* approach for the identification of isosurfaces. Specifically, it quickly identifies the locations corresponding to a particular isosurface without wasting much effort on extraneous locations. For this purpose, we leverage our basic assumptions of monotonicity and smoothness of plan costs – these imply that in each dimension of the discretized ESS, the optimal costs are in increasing order and do not change abruptly.

We begin by presenting the algorithm for a 2D ESS, followed by the extension to higher-dimensional selectivity spaces. The notations used in this section are summarized in Table 7.1.

Notation	Description
C	Cost of isosurface
res	Resolution of the ESS grid
α	Cost tolerance factor in isosurface identification
$c_{opt}(q)$	Optimal cost at location q in the ESS
L	A generic isosurface location in the ESS
$L_{x\pm1}$	ESS locations in immediate neighborhood of L along dimension x
S	Initial seed location for an isocost surface in the ESS

Table 7.1: Reference table for Notations for NEXUS algorithm

7.1.1 2D ESS

Given a location L with coordinates (x, y) in the discretized 2D ESS, we denote its three immediate "lower" neighbors as follows: $(x-1, y)$ with L_{x-1}; $(x, y-1)$ with L_{y-1}; and $(x-1, y-1)$ with L_{-1}. With these notations, the location $L(x, y)$ is included in the contour C if it satisfies the following conditions:

(a) $C \leq c_{opt}(L) \leq (1+\alpha)C$ and

(b) $c_{opt}(L_{x-1}) < C$ or $c_{opt}(L_{y-1}) < C$ or $c_{opt}(L_{-1}) < C$.

The first condition establishes the acceptable cost interval for L, while the second ensures that at least one of L's dominated neighbors is outside of the cost interval (to prevent "surface

[1]This is joint work with C.Rajmohan [79]

thickening", as explained earlier). It is to be noted here that, in condition(b) the RHS value cannot be more than C since otherwise L won't satisfy condition (a) which is the primary condition for contour acceptability.

With the above setting, the contour identification algorithm works in two phases:

1. *Locating the Initial Seed:* Here, the aim is to find the contour location that has the *maximum 'y'* coordinate, and use it as a *seed* location for the next phase of the algorithm. This extreme point can only lie on either the left edge or the top edge of the ESS, i.e. (0,0) to (0, res) or (0,res) to (res, res). To determine the correct edge, we simply cost these three ESS corners, and determine which edge includes C in its range of values. Once the edge has been determined, the exact location S, to serve as the initial seed, is determined using a *binary search* on that edge using the cost value C.

2. *Neighborhood EXploration Using Seed (NEXUS):* Since the seed has the maximum 'y' location, for locating our next isocost point, we need to only consider the 3^{rd} and 4^{th} quadrants relative to the seed as origin. However, locations in the 3^{rd} quadrant are already known to be *unacceptable* due to PCM. Therefore, the initial seed location S can be used to recursively generate new seed locations solely in the 4^{th} quadrant and thus grow the contour.

   For a given seed location $S(x, y)$, we denote the location $(x + 1, y)$ with S_{x+1} and the location $(x, y - 1)$ with S_{y-1}. By virtue of PCM, we know that $c_{opt}(S_{x+1}) > c_{opt}(S)$ and $c_{opt}(S_{y-1}) < c_{opt}(S)$. We find from the query optimizer the optimal costs for these candidate seed locations, and choose the new seed based on the following simple criterion:

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

   The end of this recursive routine is marked by the non-existence of both S_{x+1} and S_{y-1} in the ESS grid.

A sample working of the above algorithm is visually demonstrated through a set of Figures.

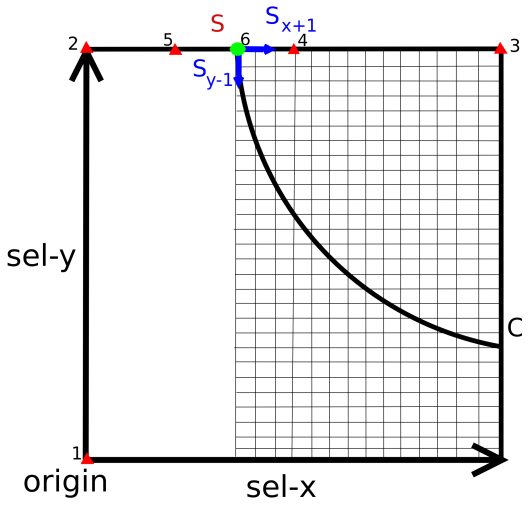


Figure 7.1: Finding seed location in 2D ESS

First, the identification of the initial seed S using 6 optimization calls is shown in Figure 7.1. Then the recursive contour exploration in the 4th quadrant of S is shown in Figure 7.2 – here, the optimized locations are marked with either a red colored triangle ▲ or a green colored dot • – the latter constitute the accepted contour locations, whereas the former indicates locations that were explored but rejected. Finally, Figure 7.3 shows the contour exploration completing when S hits the ESS boundary.

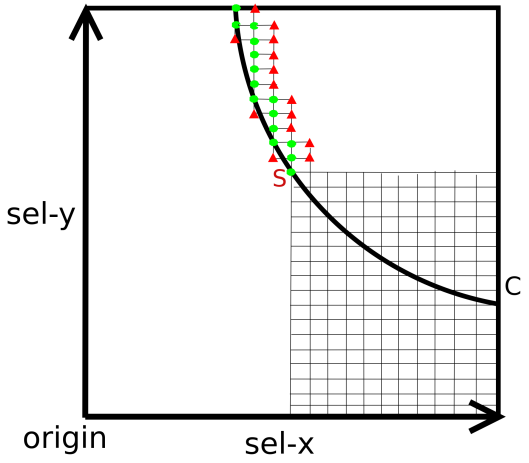


Figure 7.2: Intermediate Contour Exploration in 2D ESS

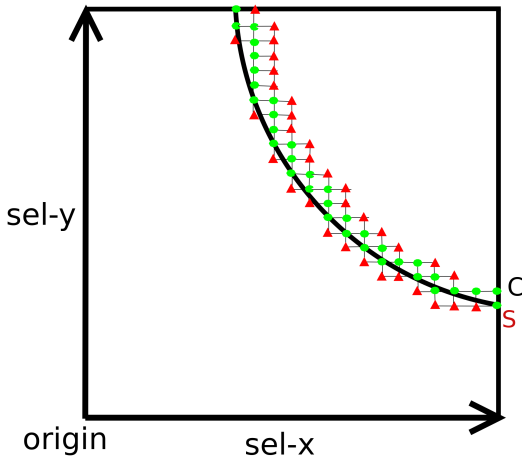


Figure 7.3: Completely Explored Contour in 2D ESS

Discussion It is noteworthy that $\#red\blacktriangle = \#green\bullet$, i.e. the algorithm performs exactly *twice* the number of optimizer calls as compared to the optimal algorithm that finds only acceptable contour locations. This is because at any point during contour exploration, there are *exactly two* candidates for the new seed – S_{x+1} and S_{y-1}, and one of them will definitely be on the (accepted) contour. In fact, it is easy to see that since the decision is based solely on S_{y-1}, the optimization call for S_{x+1} should be invoked only if required, and thereby further reduce the number of wasted optimization calls.

7.1.2 Extension to nD ESS

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

Locating the Initial 3D Seed Here, the initial seed S is the isosurface location with the maximum z coordinate. To find this point, it is first checked whether the seed lies on the edge $(0, 0, 0)$ to $(0, 0, res)$, which implies that $S = (0, 0, z)$ with $z < res$. If yes, the seed can be determined by using a binary search on this edge – this corresponds to Case 1 in Figure 7.4. If no, the initial seed is located using a procedure similar to 2D ESS for the XY-slice with

$z = res$, which is visualized as Case 2 in Figure 7.5 and 7.6 - here, there are two possibilities: $S = (0, y, res)$ with $y < res$ (Case 2a), or $S = (x, res, res)$ with $x < res$ (Case 2b).

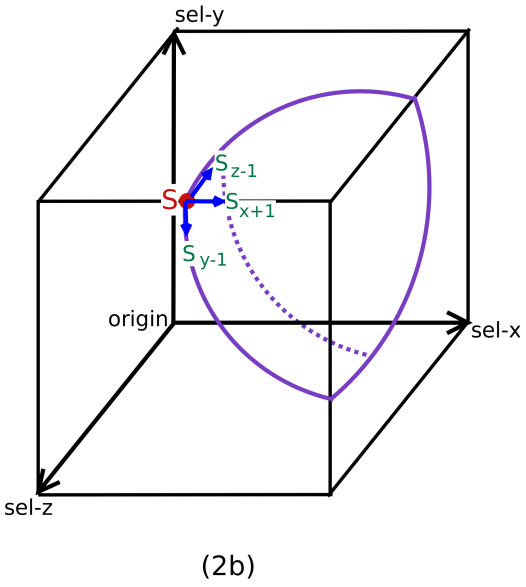
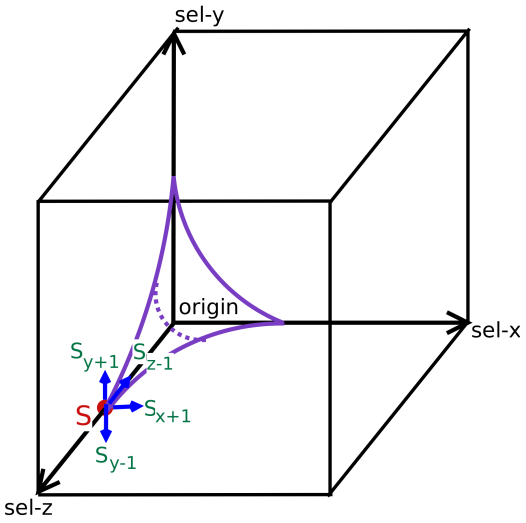


Figure 7.4: Example Contour Exploration in 3D ESS (Case 2b)

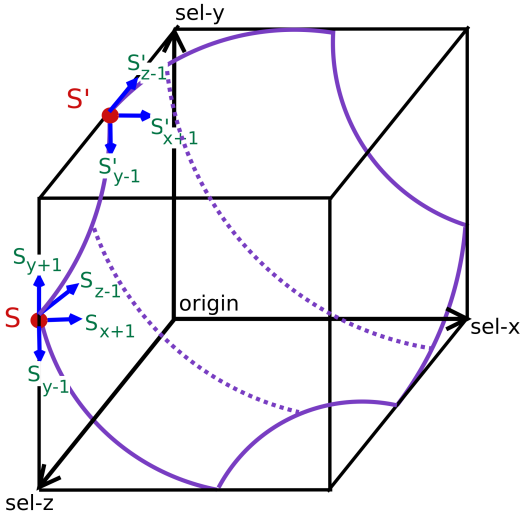
3D Isosurface Exploration We first explain the isosurface exploration phase for Case 2b. To identify all isosurface locations with $z = res$, we use the 2D exploration algorithm for the XY-slice with $z = res$, and grow the initial seed S as explained previously. For exploring the locations with lower values of z, the initial seeds for each XY-slice are generated by 2D exploration of the XZ-slice corresponding to $y = res$, using the initial seed S and candidate locations S_{x+1} and S_{z-1}.

Similarly in Case 1, the initial seeds for each lower value of z are generated by exploring the YZ-slice corresponding to $x = 0$, starting with an initial seed S and candidate locations S_{y+1} and S_{z-1}. Finally, the algorithm for Case 2a proceeds in two sub-phases where the first sub-phase is similar to Case 1 until it finds a seed with $y = res$ (shown as S' in Case 2a of Figure 7.5), and thereafter in the second phase it follows an algorithm similar to Case 2b of of Figure 7.6).



(1)

Figure 7.5: Example Contour Exploration in 3D ESS (Case 1)



(2a)

Figure 7.6: Example Contour Exploration in 3D ESS (Case 2a)

Generic nD Algorithm In D-dimensional space, the initial seed location is of the form $(0^s, v, res^t)$ where $0 < v < res$ and $s + t = D - 1$. Given such a seed, the dimension-pair (d_{s+1}, d_{s+1+t}) is used to generate more seeds through the 2D algorithm, and for each such seed, the $D - 1$ dimensional subproblem over the dimensions $(d_1, d_2, \ldots, d_{s+t})$ is recursively

solved. The recursion terminates with the completion of 2D exploration of the dimension-pair (d_{s+1}, d_{s+1+t}).

7.1.3 Impact on Bouquet Identification Overheads

Overall, NEXUS can be used to either: (a) enable an early-start for the bouquet execution phase without invoking CSI; or, alternatively (b) reduce the total effort of identifying all isosurface plans before using CSI (by ignoring the ESS regions that lie in between the isosurfaces). In addition, this approach also makes isosurface exploration a highly parallelizable task since in principle, a new thread can be created whenever a seed is generated for a lower dimensional subspace.

7.2 Implementing Plan Swallowing

As discussed in Section 6.1, the natively high plan-density values for the isosurfaces in ESS can be reduced by reducing the size of POSP itself, by utilizing the plan-swallowing enhancement. While the enhancement is usually quite effective in terms of reduction in cardinality of POSP set but is computationally intensive as it requires: (a) optimization call for each location in ESS to compute POSP and (b) costing calls across the ESS for each plan in POSP to get its behavior across space, to apply the enhancement using CostGreedy-FPC routine [47].

But in the bouquet approach, requirement is to minimize the maximum plan density across the isosurfaces rather than finding the smallest set of plans for the entire ESS. For this reason, we utilize the plan-swallowing enhancement separately for each of the isosurfaces. Further, while it is best to use the entire POSP as the swallower set to get maximum reduction for individual isosurfaces. We make use of another efficient variant, termed as *intra-surface* plan swallowing, where the swallower set contains only the plans from the same isosurface. In principle, it is possible that the intra-surface alternative fairs badly in terms of its ability to reduce the plan-density of the isosurface. But, since it was emprically shown in [79] that the reduction quality does not degrade noticeably, we use intra-surface variant of plan swallowing enhancement. An

additional advantage with intra-surface processing is that it fits well with the parallel processing capability of NEXUS, which means that the isosurfaces with reduced effective plan-density can be produced independent of one another.

7.3 Implementing Execution Covering

7.3.1 Covering Sequence Identification Algorithm

Clearly, the basic bouquet algorithm provides a simple to implement solution where no skipping of executions take place. On the other hand, the exhaustive algorithm enumerates all candidate covering sequences, but incurs a computational effort that is exponential in the number of executions present in the bouquet sequence. To bridge this gulf, we describe here a polynomial-time algorithm, CSI, that attempts to improve the MSO_g of the bouquet sequence with the following idea – "find covering executions for plans on the MSO causing contour and its predecessors".

For instance, the MSO_g of 24.5 in the example of Figure 6.3b is caused at IC_4 due to the aggregate impact of $\omega(CS^1)$ through $\omega(CS^4)$. This MSO_g can be improved by finding the covering sequences individually for the contours IC_1 through IC_4. For this purpose, we describe below a subroutine of CSI that finds the covering sequence for a contour IC_k while trying to minimize its effective plan density – employing this routine brings the MSO_g in Figure 6.3b down to 14.5 .

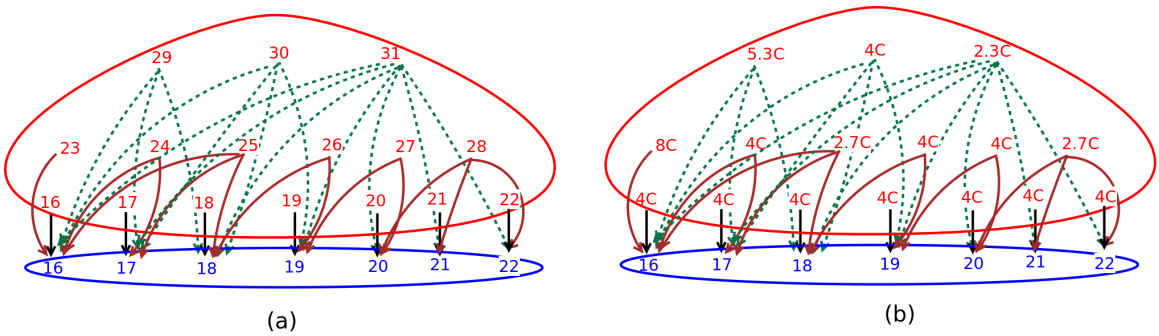


Figure 7.7: Adapting Red-Blue Weighted Domination for solving *minimize*$(w(CS^4))$

Minimizing $\omega(CS^k)$ The problem of minimizing $\omega(CS^k)$ can be abstracted as an adaptation of the *Red Blue Weighted Dominating Set* [37] problem - a sample adaption corresponding to our running example is shown in Figure 7.7a.

Here, the blue set contains nodes from contour IC_k. and the red set contains nodes from the contours IC_k through $IC_{k'}$, where $cost(IC_{k'}) \leq |IC_k| \times cost(IC_k)$. Apart from the covering edges (solid brown) borrowed from the Hasse diagram (i.e., IC_{k+1} to IC_k), it also includes transitive edges (dotted green) for farther contours (IC_{k+2} onwards) and identity edges (solid black) from red version to blue version of IC_k nodes. Finally, the weight of any red node is given by the corresponding value of $\omega(E_i)$.

With the above modeling, the problem of minimizing $\omega(CS^k)$ is the same as finding the minimum-weight subset of red nodes that can dominate all the nodes in the blue set. Now, since red-blue weighted domination is known to be a NP-hard problem, and also equivalent to the Minimum-weight Set-Cover problem [37], we have utilized an adaptation of the *greedy weighted set-cover* algorithm to ensure efficiency – the greedy criterion is the minimum weight per newly covered blue node. The pseudocode for the resulting subroutine is shown in Algorithm 3.

For the example formulation in Figure 7.7a, the weight per covered blue node for each of the red nodes, is shown in Figure 7.7b. Here we find that E_{31} is the greedy red choice, and it is the solution since there are no more blue nodes to be covered. The greedy subroutine has run time complexity of $V_{blue}log(V_{red})$, and approximation factor of $log(V_{blue}) + 1$. The covering sequence obtained for IC_4 with this approach is shown in Figure 7.8.

The CSI Algorithm CSI begins by identifying the contour that causes the $\mathsf{MSO_g}$, denoted IC_{k^*}, and then applies the minimization subroutine on contours IC_1 through IC_{k^*} in sequence. After this pass, it is possible that the "culprit" $\mathsf{MSO_g}$ contour has now shifted to a new contour $IC_{k^{**}} > IC_{k^*}$ – if so, another minimization pass is carried out for contours from IC_{k^*} through $IC_{k^{**}}$. Otherwise, the algorithm is concluded since $\mathsf{MSO_g}$ cannot be improved further.

The motivation for processing contours in increasing cost order during each minimization

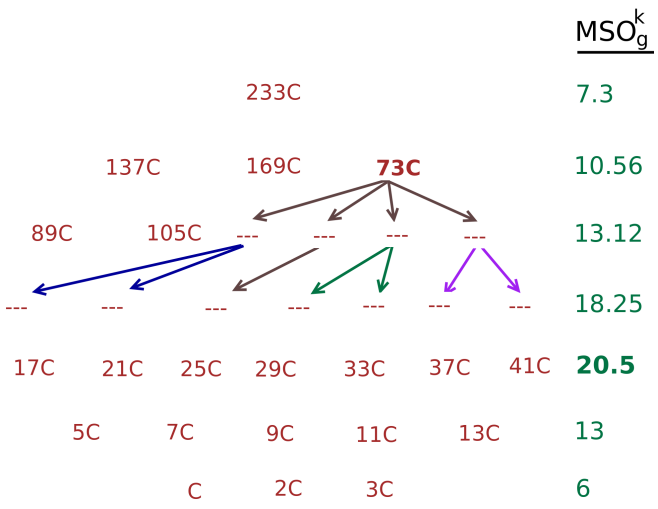


Figure 7.8: Solution for $minimize(w(CS^4))$ using red-blue domination

pass is that the covering nodes identified for contour IC_k may also cover nodes from contours IC_{k+1} and beyond, thereby reducing the number of uncovered nodes for higher contours.

Algorithm 3: Minimize $\omega(CS^k)$ Subroutine

Input: $V_{blue} = IC_k, V_{red} = (IC_k \cup \ldots \cup IC_{k'}), \omega : BS \rightarrow \mathbb{R}$
Output: V_{cov}
// V_{cov} is the set of covering executions (subset of V_{red})
$V_{cov} = \{\}$;
// V_{dom} is the set of executions covered by V_{cov} (subset of V_{blue})
$V_{dom} = \{\}$;
// for each node v in V_{red}, $dom(v)$ denotes the set of executions that it can cover in V_{blue}
while $V_{dom} \subset V_{blue}$ **do**
  $v_{sel} = \phi$;
  $f(v_{sel}) = \infty$;
  for $v \in V_{red} \setminus V_{cov}$ **do**
    $f(v) = \frac{\omega(v)}{|dom(v) \setminus V_{dom}|}$;
    if $f(v_{sel}) > f(v)$ **then**
      $v_{sel} = v$;
    end
  end
  $V_{dom} = V_{dom} \cup dom(v_{sel})$;
  $V_{cov} = V_{cov} \cup \{v_{sel}\}$;
end

7.3.2 Efficiently Constructing Hasse Diagram of Executions

In principle, the CSI routine needs to determine the cover relation among *all pairs* of executions in BS. However, the explicit checking of these pairs can be reduced by leveraging the following inference tests:

1. If both executions are from the same contour, reject the pair as a cover relation cannot exist between them.

2. If one of the executions is $E_{terminal}$, then accept the pair since the cover exists, by definition of $E_{terminal}$.

3. If the executions lie more than one contour apart, then explicit evaluation is required only if a transitive cover relation is non-existent.

Even with the above pruning, the processing required for the remaining pairs may still turn out to be computationally significant, since the explicit test for a pair (E_i, E_j) is equivalent to establishing whether $R(E_j)$ is a subset of $R(E_i)$. We therefore present next an alternative procedure to determine the cover relation $E_i \succeq_{cover} E_j$, which does not entail this subset check.

Cover Determination Procedure

Let $L(E_i)$ denote the contour locations for E_i. We use MAX_i to represent the ESS location whose value on each dimension corresponds to the maximum selectivity coordinate on that dimension across all contour locations of E_i. That is,

$$s_d(MAX_i) = \max_{l \in L(E_i)} s_d(l) \text{ for } 1 \leq d \leq D$$

Similarly, we denote with MIN_i the ESS location corresponding to the minimum selectivity

coordinate for each dimension across all contour locations of E_i. That is,

$$s_d(MIN_i) = \min_{l \in L(E_i)} s_d(l) \text{ for } 1 \leq d \leq D$$

To make these notions concrete, the MAX_i and MIN_i locations for the first 3 contours of the example ESS are shown as *red* squares (■) and *blue* diamonds (♦), respectively, in Figure 7.9.

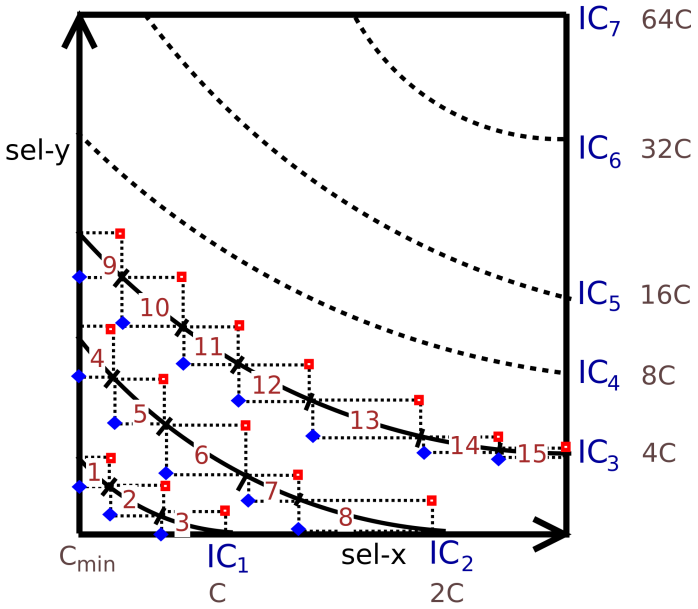


Figure 7.9: MAX_i and MIN_i locations for executions on contours IC_1 to IC_3

To determine the cover existence between E_i and E_j, we first employ two heuristic checks that may not be decisive in all cases, but are quite efficient to evaluate. Their efficiency stems from their usage of simple *product order*[1] based checks among only MAX_i's and MIN_i's, which are easily computable through a single scan of the $L(E_i)$ locations.

Product Check 1: The necessary and sufficient condition that $MIN_i \succeq_{product} MAX_j$ for $E_i \succeq_{cover} E_j$, helps to efficiently identify true positives. For example, it can quickly

[1]We use generalized version of the standard product order which is defined as: Given two pairs (a_1,b_1) and (a_2, b_2) in $A \times B$, one sets (a_1, b_1) $\succeq_{product}$ (a_2, b_2) if and only if $a_1 \geq a_2$ and $b_1 \geq b_2$.

determine that in Figure 7.9, the candidate execution pair (E_7, E_3) satisfies the cover relation.

Product Check 2: The necessary condition that $MAX_i \succeq_{product} MAX_j$ for $E_i \succeq_{cover} E_j$, helps to quickly reject true negatives. For instance, (E_5, E_3) can be quickly rejected using this criterion.

In the event that the above two checks are not conclusive, we employ a final check that is decisive in all scenarios, but is comparatively expensive since it requires, for each location in set $L(E_i)$, processing the *entire set* of locations in $L(E_j)$.

Product Check 3: The condition $L(E_i) \succeq_{product} L(E_j)$ is sufficient to decide whether $E_i \succeq_{cover} E_j$. Here, $L(E_i) \succeq_{product} L(E_j)$ is satisfied if there exists a location $l^s \in L(E_i)$ for each $l^t \in L(E_j)$, such that $l^s \succeq_{product} l^t$.

While certainly more expensive than Checks 1 and 2, note that Check 3 is expected to be relatively more efficient than the direct region-subset check, since typically $|L(E_i)| \ll |R(E_i)|$.

To highlight the potency of the above evaluation procedure, consider for instance the neighboring contours IC_2 and IC_3 in Figure 7.9. Here, there are 35 execution pairs arising from the contours, and among them, 6 pairs are identified as true positives (Check 1), 21 pairs are rejected as true negatives (Check 2), leaving only 8 pairs for the final comparison (Check 3).

Chapter 8

Plan Bouquet Architecture and Prototype Implementation

8.1 Bouquet Architecture and Essential API Features

For a given Q with an ESS, the query execution workflow of the bouquet approach becomes operational as shown in Figure 8.1. For this purpose, the database engine needs to support the following functionalities: (1) selectivity injection; (2) abstract plan costing and execution; and (3) cost-budgeted partial execution of plans. Next, we elaborate on the usage of each of these features, followed by implementation details of the bouquet driver layer.

8.1.1 Selectivity Injection

For isosurface exploration using the algorithm described in Section 7.1, we need to be able to systematically generate queries with the desired ESS selectivities. One option is to, for each new location, suitably modify the query constants and the data distributions, but this is clearly highly cumbersome and time-consuming. We have therefore taken an alternative approach in our PostgreSQL implementation, wherein the optimizer is instrumented to directly support *injection* of selectivity values in the cost model computations. Interestingly, some

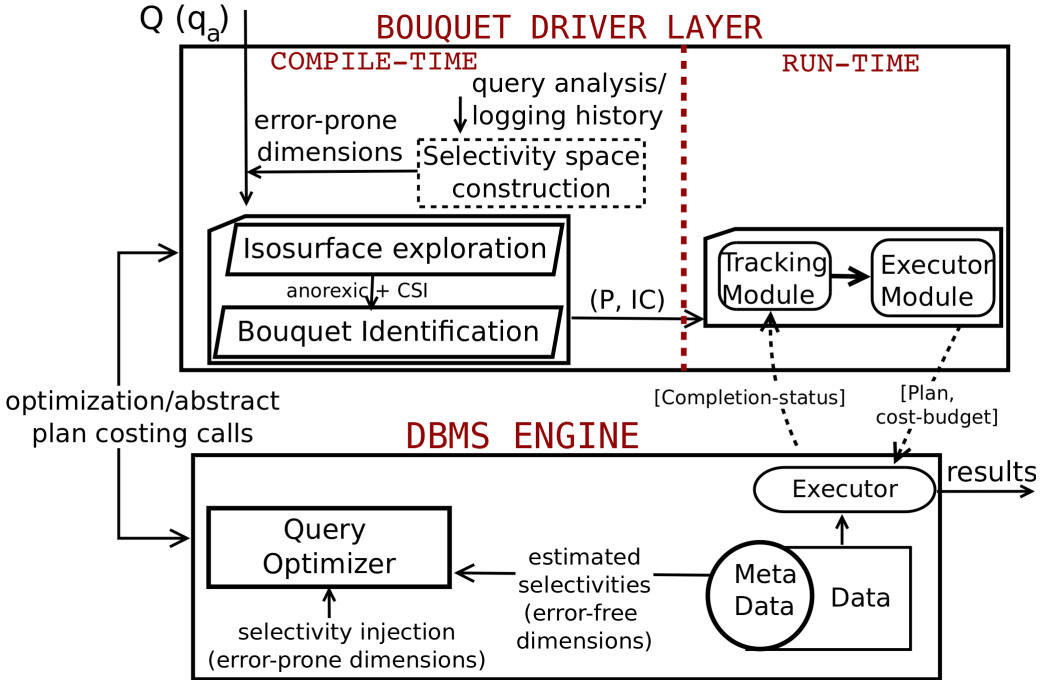


Figure 8.1: Architecture of Bouquet Mechanism

commercial optimizer APIs already support such selectivity injections to a limited extent (e.g. IBM DB2 [4]).

8.1.2 Abstract Plan Costing and Execution

Once the isosurfaces have been explored, we need to reduce their plan densities using the anorexic reduction technique, as explained in Section 6.1. This is achieved through the FPC variant of the Cost Greedy algorithm [46], which requires an abstract plan costing feature for estimating the cost of a plan outside its optimality region. This feature is already supported by some commercial optimizers (e.g. [5]).

Further, during the bouquet execution phase, we need to be able to instruct the execution engine to execute a particular bouquet plan. This feature also is currently provided by a few commercial systems (e.g. [5]).

8.1.3 Cost-budgeted Execution

The bouquet approach requires, in principle, only a simple "timer" that keeps track of elapsed time, and terminates plan executions if they exceed their assigned cost budgets. No material changes need to be made in the engine internals to support this feature. The premature termination of plans can be achieved easily using the *statement.cancel()* functionality supported by JDBC drivers. Note that although bouquet identification provides budgets in terms of abstract optimizer cost units, they can be converted to equivalent time budgets through the techniques proposed in [87].

8.1.4 Bouquet Driver Layer

As highlighted in Figure 8.1, we have an external program, the "Bouquet Driver", which treats the query optimizer and executor as black-boxes. First, it interacts with the query optimizer module to determine the isosurfaces and the plan bouquet. Then it performs executions of the bouquet plans using an *execution* client and a *tracking* client. The execution client selects the plan to be executed next, while the tracking client keeps track of the time elapsed, and terminates the execution if the allotted time budget is exhausted.

8.2 QUEST Prototype

This prototype implementation of the bouquet technique[1] helps an interested user to *visually* observe the selectivity estimation problems that plague current database optimizers, and the novel robustness characteristics that the bouquet technique brings to bear on these chronic problems. A two-dimensional ESS query[2] based on Query 5 of the TPC-H benchmark, operating on a fully indexed 1 GB standard TPC-H database hosted on the PostgreSQL engine, will be used as a running example to explain these scenarios.

[1] This is joint work with Sumit Neelam [71].
[2] With *customer* and *lineitem* as error-prone selectivity dimensions.

8.2.1 Sub-optimality of Native Optimizer

The first screen highlights the estimation errors and its impact on plan choice and execution performance for the input query. A sample instance of the corresponding QUEST interface is shown in Figure 8.2, where the user can observe:

- An operator-level comparison between $P_{opt}(q_e)$ and $P_{opt}(q_a)$ – in this instance, $P_{opt}(q_e)$ features a series of *Nested Loop* joins while $P_{opt}(q_a)$ opts for *Hash Joins*, and the join orders are different.
- The locations of q_a and q_e in the ESS, and the large error gap between them – in this instance, q_a=(30.9%, 26.7%) while q_e is underestimated to be (0.25%, 3.1%).
- The adverse performance impact due to the estimation error – in this instance, the sub-optimality is around 17.

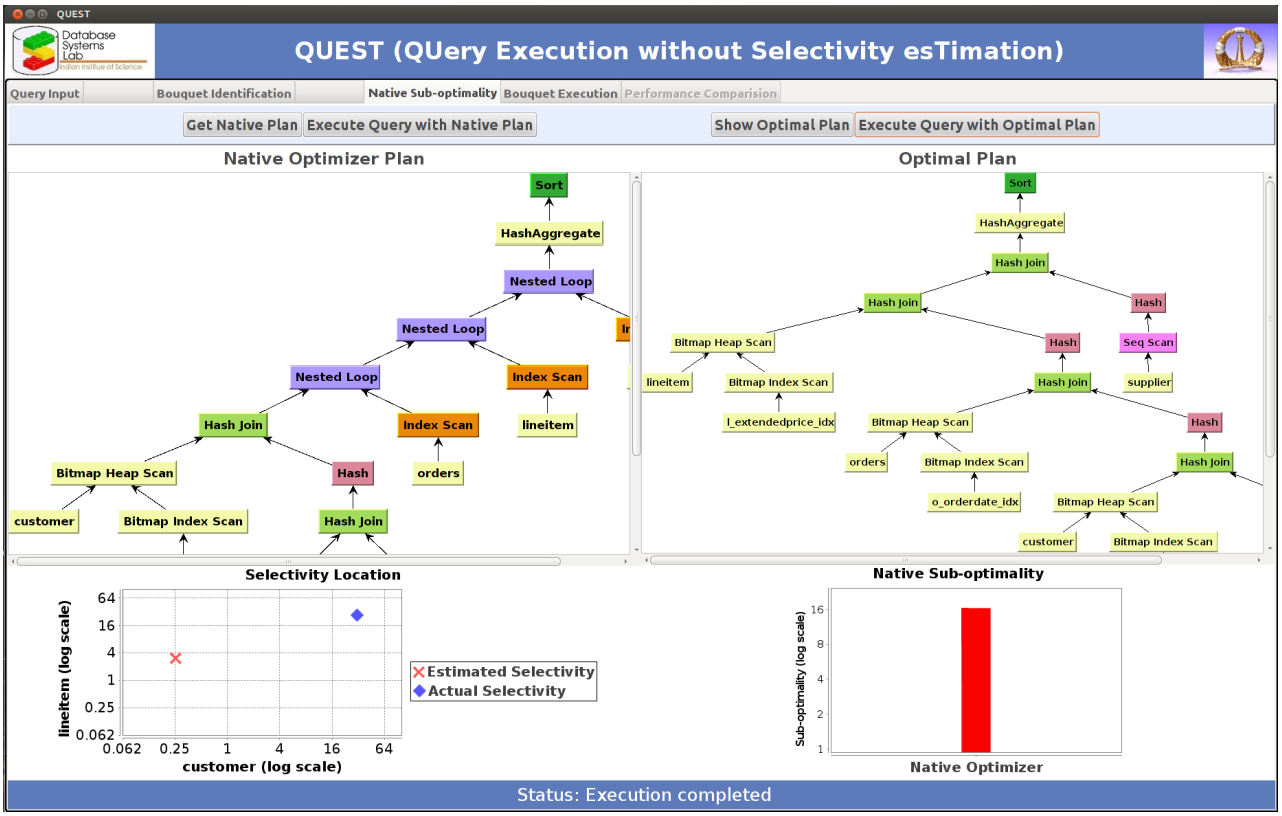


Figure 8.2: Sub-optimality of Native Optimizer

8.2.2 Bouquet Identification

Turning our attention to bouquet technique, we start with a description of the compile-time phase i.e. bouquet identification, whose graphical display 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 *plan swallowing enhancement* [46] in order to reduce the number of plans to a small number without substantively affecting the query processing quality (cost-increase threshold λ) of any individual query in the selectivity space. On this reduced diagram the bouquet's distinctive feature of cost-based discretization using geometrically increasing isocost planes (common ratio r) is applied – the combined effect of reduction (λ) and discretization (r) 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 plan swallowing enhancement[1], the plan cardinality goes down to 6 plans. Finally, the PIC is divided using 5 isocost contours with r=2, and the POSP plan cardinality distribution on these contours is (4, 4, 4, 3, 1).

Here, the user will be able to provides values of their choice for reduction parameter λ and discretization parameter r. For these values, the resulting MSO guarantee will be evaluated and compared against the MSO guarantee with recommended values of the parameters ($\lambda = 20\%, r = 2$).

8.2.3 Bouquet Execution

The next step illustrates execution phase of the bouquet technique as calibrated sequence of budgeted 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

[1]With reduction parameter $\lambda = 20\%$, as recommended in [46].

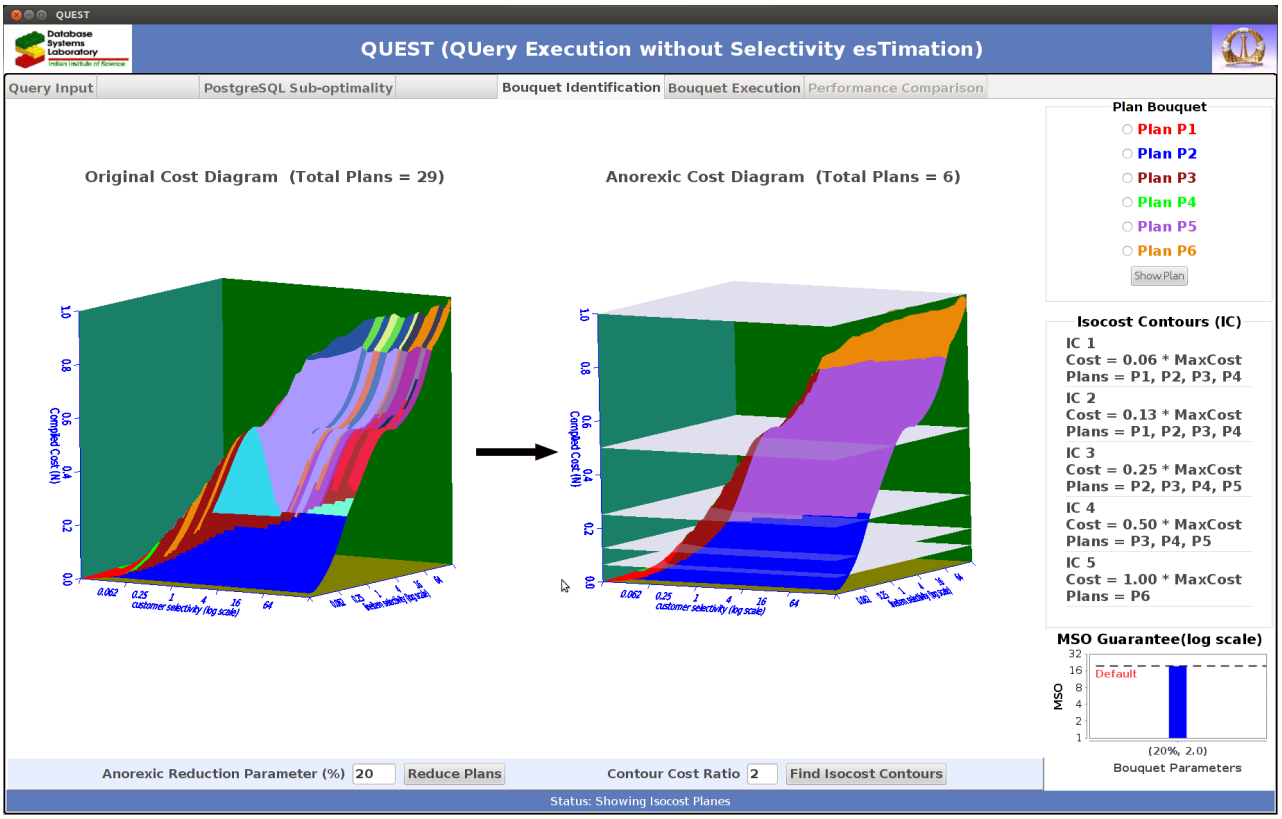


Figure 8.3: Bouquet Identification Interface

to completion within its assigned budget. The dynamism of this iterative process is captured in the interface shown in Figure 8.4, which is continually updated to show:

- The ESS region covered by each partial plan execution – subsequent to each such execution, the associated region is shadowed with the plan's color.
- The execution order timeline of the plans, along with their tree structures – this allows database analysts to carry out offline replays of the plan execution sequence.
- The contour budgets, which initially appear as white bars of geometrically increasing height, 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 (for the sample query, it is around 3.7).

Here, 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.

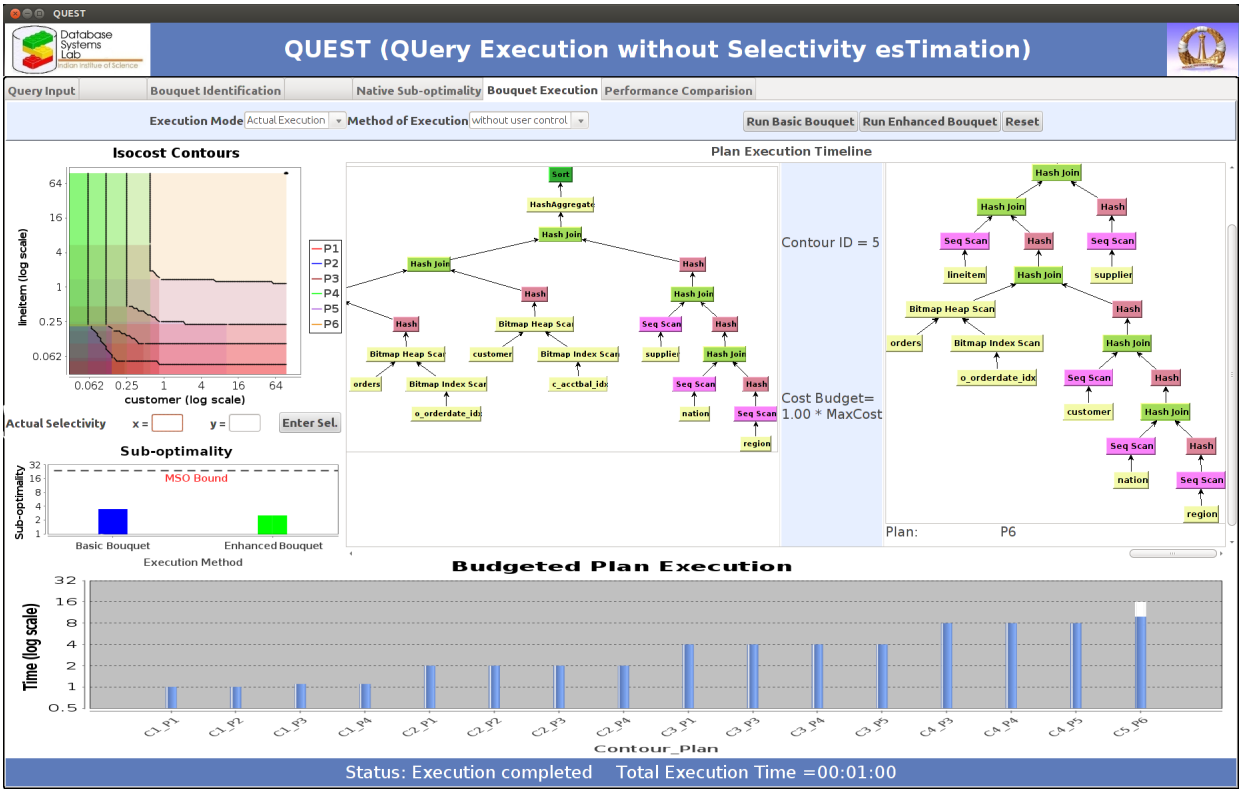


Figure 8.4: Bouquet Execution Interface

8.2.4 MSO Guarantees & Repeatability

The interface also provides the user with an opportunity to verify for themselves the MSO and repeatability guarantees offered by the bouquet technique. Firstly, with regard to the MSO guarantee, user can fill in any desired location of q_a in the text box shown in Figure 8.4 (below the isocost contours), and then invoke the bouquet algorithm on this query instance to confirm that the sub-optimality incurred is within the apriori stated bound (for the sample query, this MSO bound is less than 20, which is orders of magnitude lower than the empirically determined MSO of 10^4 obtained with the native optimizer).

Secondly, with regard to repeatability, our goal is to show that, unlike the native optimizer,

the bouquet execution sequence is only a function of q_a, and not of q_e. To this end, we can radically alter, using the CODD metadata editing tool [31], the distribution histograms of the attributes featured in the query, while keeping the underlying data *unchanged*. Subsequent to the alteration, the bouquet algorithm can be re-executed and confirmed to behave identically to its prior incarnation.

Chapter 9

Empirical Evaluation

We now turn our attention towards profiling the performance of the bouquet approach on a variety of complex OLAP queries, using the MSO, ASO and MH metrics enumerated in Chapter 3. In addition, we also describe experiments that show: (a) spatial distribution of robustness in the ESS; (b) low bouquet cardinalities; (c) low sensitivity of the MSO_g to the λ reduction parameter; and (d) extension of the results to commercial databases. As specified in Chapter 3, the entire evaluation is carried out using optimizer costs, while assuming that all combinations of the actual and estimated query locations are possible in the ESS.

Before going into the evaluation details, we describe the experimental setup and the rationale behind the choice of comparative techniques. This is followed by a brief discussion on the compile-time overheads incurred by the bouquet algorithm.

9.1 Experimental Setup

Database Environment The test queries are chosen from the TPC-H and TPC-DS benchmarks to cover a spectrum of join-graph geometries, including *chain*, *star*, *branch*, etc. with the number of base relations ranging from 4 to 8. The number of error-prone selectivities range from 3 to 5 in these queries, all corresponding to join-selectivity errors, for making challenging multi-dimensional ESS spaces. We experiment with the TPC-H and TPC-DS databases at

their default sizes of 1GB and 100GB, respectively. Finally, the physical schema has indexes on all columns featuring in the queries, thereby maximizing the cost gradient $\frac{C_{max}}{C_{min}}$ and creating "hard-nut" environments for achieving robustness.

The summary query workload specifications are given in Table 9.1 – the naming nomenclature for the queries is xD_y_Qz, where x specifies the number of 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 selectivity space on Query 5 of the TPC-H benchmark.

Query	**Join-graph (# relations)**	$\frac{C_{max}}{C_{min}}$	**Query**	**Join-graph (# relations)**	$\frac{C_{max}}{C_{min}}$
3D_H_Q5	chain(6)	16	3D_DS_Q96	star(4)	185
3D_H_Q7	chain(6)	5	4D_DS_Q7	star(5)	283
4D_H_Q8	branch(8)	28	4D_DS_Q26	star(5)	341
5D_H_Q7	chain(6)	50	4D_DS_Q91	branch(7)	149
3D_DS_Q15	chain(4)	668	5D_DS_Q19	branch(6)	183

Table 9.1: Query workload specifications

System Environment For the most part, the database engine used in our experiments is PostgreSQL 8.4 [3], equipped with the API features described in Section 8.1[1]. Specifically, the first two features were introduced with minimal changes to the source code. On the other hand, cost-budgeted execution is natively supported by invoking the following command at the tracking client: "*select pg_cancel_backend(process_id)*". The required *process_id* (of the execution client) can be found in the view *pg_stat_activity*, which is maintained by the engine itself.

The hardware platform is a vanilla Sun Ultra 24 workstation with 8 GB memory and 1.2 TB of hard disk.

Comparative Techniques In the remainder of this section, we compare the bouquet algorithm (with anorexic parameter $\lambda = 20\%$ and CSI enhancements) against the native PostgreSQL

[1]All the experiments have been repeated on PostgreSQL 9.4 as well with virtually no difference in the results.

optimizer, and the SEER robust plan selection algorithm [47].

SEER uses a mathematical model of plan cost behavior in conjunction with anorexic reduction to provide replacement plans ($P_{rep}(q_e)$ for q_e) that, at all locations in the ESS, either improve on the native optimizer's performance, or are worse by at most the λ factor. It is important to note here that, in the SEER framework, the comparative yardstick is $P_{opt}(q_e)$, the optimal plan at the *estimated* location, whereas in our work, the comparison is with $P_{opt}(q_a)$, the optimal plan at the *actual* location. Still, it has been shown in [47] that in many cases $c(P_{rep}(q_e), q_a) \ll c(P_{opt}(q_e), q_a)$, hence SEER is expected to perform better than the native optimizer on our sub-optimality based metrics. Finally, since $c(P_{rep}(q_e), q_a) \leq (1 + \lambda) \times c(P_{opt}(q_e), q_a) \ \ \forall q_a \in ESS$, we can infer that MH $\leq \lambda$ with SEER.

On the other hand, purely heuristic-based reoptimization techniques, such as POP [65] and Rio [12], are not included in the evaluation suite. No doubt they can be very effective when the estimation errors are small in magnitude and number. But when the errors are significant, as is commonplace in practice [62], their performance on our metrics (MSO or MH) could be *arbitrarily poor*. This is because of their inability to provide worst-case performance guarantees – in fact, they are unable to do so with regard to both $P_{opt}(q_e)$ and $P_{opt}(q_a)$, as explained in detail in Appendix.

Further, the heuristics that POP and Rio employ are more appropriate for low-dimensional spaces – for example, that near-optimality of a plan at the corners of the principal diagonal of the error space, implies near-optimality in the interior of this space; or, that the selectivity validity ranges found by comparing only with the class of structure-equivalent plans, provide good approximations to the true ranges. Therefore, these techniques may not work well when faced with large multi-dimensional estimation errors, which is the primary target of our work.

For ease of exposition, we will hereafter refer to the bouquet algorithm, the native optimizer, and the SEER algorithm as **BOU**, **NAT** and **SEER**, respectively, in presenting the results.

9.2 Compile-time Overheads

The computationally expensive aspect of BOU's compile-time phase is the identification of the plans on the isosurfaces of the ESS. For this task, we have proposed a new algorithm NEXUS, as described in Section 7.1, that can selectively explore locations for a particular isosurface, and ignore the remaining portion of the ESS. Currently, using only *single-core* processing, the compile-time overheads for 3D, 4D and 5D queries with NEXUS are typically a few minutes, a few hours, and several hours, respectively. Although the overheads continue to increase exponentially with the number of ESS dimensions, we wish to highlight that NEXUS is highly parallelizable and can therefore exploit modern multi-core architectures to substantively ameliorate, in absolute terms, these overheads.

Further, as mentioned in Section 7.1, the plan bouquet execution can be overlapped with its compilation – specifically, execution can be started as soon as the first plan on the first isosurface is identified, and the identification of subsequent plans can be carried out concurrently with the ongoing executions. Finally, note that the isosurface identification process is a one-time exercise, and its overhead can be amortized by repeated invocations of the same query, which often happens with "canned" form-based interfaces in the enterprise domain.

9.3 Empirical Worst-case Performance (MSO)

In Figure 9.1, the empirical MSO performance is profiled, on a log scale, for a set of 10 representative queries submitted to NAT, SEER and BOU. The first point to note is that NAT is *not* inherently robust – to the contrary, its MSO is huge, ranging from around 10^3 to 10^7. Secondly, SEER also does not provide any material improvement on NAT – this may seem paradoxical at first glance, but is only to be expected once we realize that not *all* the highly sub-optimal (q_e, q_a) combinations in NAT were necessarily helped in the SEER framework. Finally, and in marked contrast, BOU provides *orders of magnitude* improvements over NAT and SEER – as

a case in point, for 5D_DS_Q19, BOU drives MSO down from 10^6 to around just 10. In fact, even in absolute terms, it consistently provides an MSO of *less than ten* across all the queries.

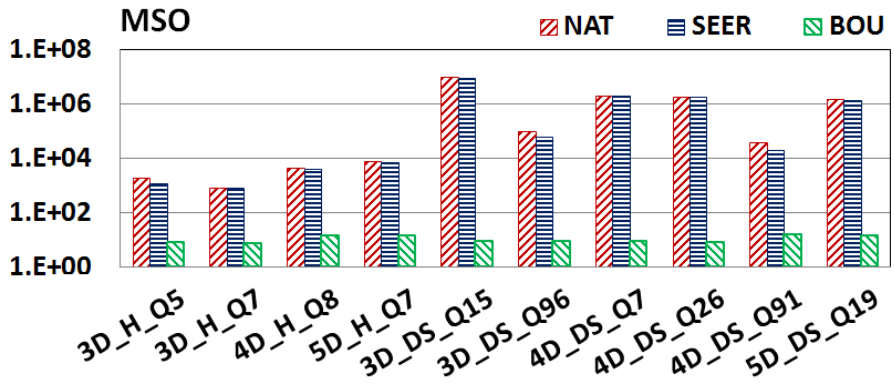


Figure 9.1: Empirical MSO Performance

9.4 Average-case Performance (ASO)

At first glance, it may be surmised that BOU's dramatic improvement in worst-case behavior is purchased through a corresponding deterioration of average-case performance. To quantitatively demonstrate that this is not so, we evaluate ASO for NAT, SEER and BOU in Figure 9.2, again on a log scale. We see here that for some queries (e.g. 3D_DS_Q15), ASO of BOU is much better than that of NAT, while for the remainder (e.g. 4D_H_Q8), the performance is comparable. Even more gratifyingly, the ASO in absolute terms is typically less than 5 for BOU. On the other hand, SEER's performance is again similar to that of NAT – this is an outcome of the high dimensionality of the ESS which makes it extremely difficult to find universally safe replacements that are also substantively beneficial.

9.5 Spatial Distribution of Robustness

We now profile for a sample query, namely 5D_DS_Q19, the percentage of locations for which BOU has a specific range of improvement over NAT. That is, the *spatial distribution* of enhanced robustness, $\frac{SubOpt_{worst}(q_a)}{SubOpt(*, q_a)}$. This statistic is shown in Figure 9.3, where we find that for the

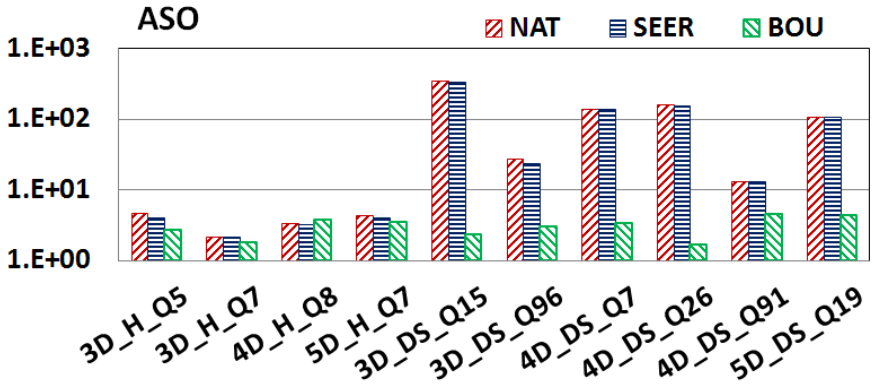


Figure 9.2: ASO Performance

vast majority of locations (close to 85%), BOU provides *two or more orders of magnitude improvement* with respect to NAT. SEER, on the other hand, provides significant improvement over NAT for specific (q_e, q_a) combinations, but may not materially help the *worst-case* instance for each q_a. Therefore, we find that its robustness enhancement is less than 10 at all locations in the ESS.

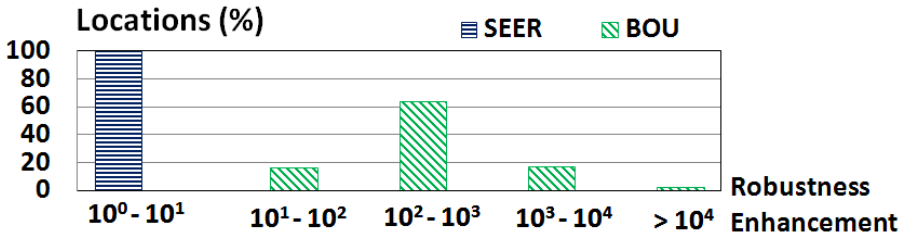


Figure 9.3: Distribution of enhanced Robustness (5D_DS_Q19)

9.6 Adverse Impact of Bouquet (MH)

Thus far, we have presented the improvements due to BOU. However, as highlighted in Chapter 3, there may be *individual* q_a locations where BOU performs poorer than NAT's worst-case, i.e. $SubOpt(*, q_a) > SubOpt_{worst}(q_a)$. This aspect is quantified in Figure 9.4 where the maximum harm is shown (on a linear scale) for our query test suite. We observe that BOU may be upto a factor of 2 worse than NAT. Moreover, SEER now steals a march over BOU since

it *guarantees* that MH never exceeds λ (= 0.2). However, the important point to note is that the percentage of locations for which harm is incurred by BOU is less than 1% of the space. Therefore, from an overall perspective, the likelihood of BOU adversely impacting performance is rare. Further, even in these few cases the harm is limited ($\leq$ MSO-1), especially when viewed against the order of magnitude improvements achieved in the beneficial scenarios.

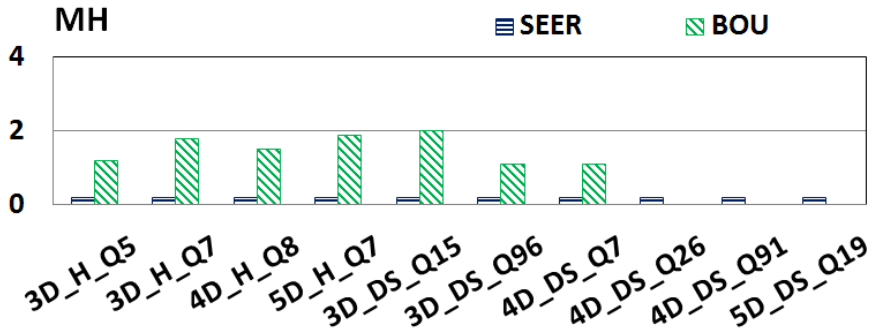


Figure 9.4: MaxHarm performance

9.7 Plan Cardinalities

The plan cardinalities of NAT, SEER and BOU are shown on a log-scale in Figure 9.5. We observe here that although the original POSP cardinality may be in the several tens or hundreds, the number of plans in SEER is orders of magnitude lower, and those retained in BOU is even smaller – only around 10 or fewer, even for the 5D queries. This is primarily due to the initial anorexic reduction and the subsequent confinement to the isosurfaces. The important implication of these statistics is that the bouquet size is, to the first degree of approximation, effectively *independent of the dimensionality and complexity of the error space.*

9.8 Commercial Database Engine

All the results presented thus far were obtained on our instrumented PostgreSQL engine. We now present sample evaluations on a popular commercial engine, hereafter referred to as COM. Since COM's API does not directly support injection of selectivities, we constructed queries

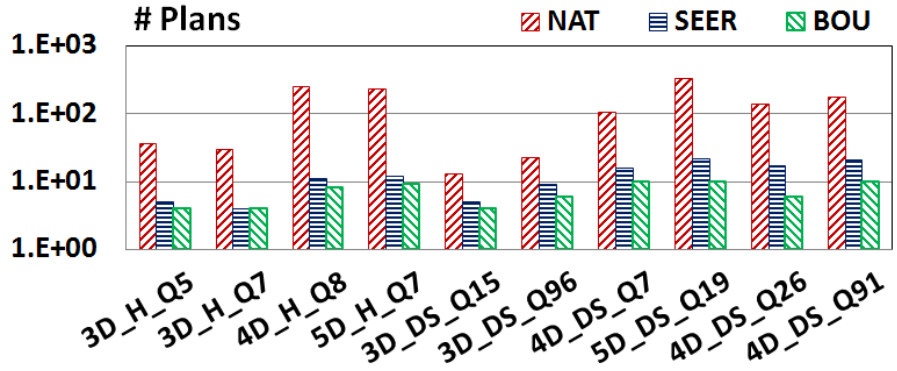


Figure 9.5: Bouquet Cardinality

3D_H_Q5b and 4D_H_Q8b wherein all error dimensions correspond to selection predicates on the base relations – the selectivities on such dimensions can be indirectly set up through changing only the constants in the query. The database and system environment remained identical to that of the PostgreSQL experiments.

Focusing on the performance aspects, shown in Figure 9.6, we find that here also large values of MSO and ASO are obtained for NAT and SEER. Further, BOU continues to provide substantial improvements on these metrics with a small sized bouquet. Again, the robustness enhancement is at least an order of magnitude for more than 90% of the query locations, without incurring any harm at the remaining locations (MH < 0). These results imply that our earlier observations are not artifacts of a specific engine.

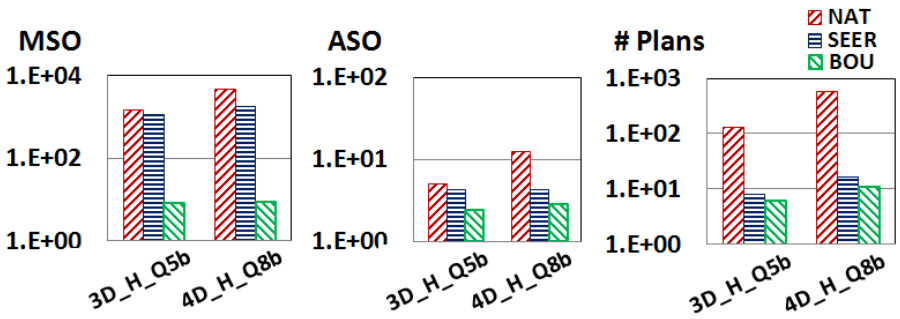


Figure 9.6: Commercial Engine Performance (log-scale)

9.9 MSO_g Sensitivity to λ Setting

Until now, both BOU and SEER were evaluated empirically over different metrics, by setting the reduction-parameter λ to 20%, a value that had been found in [46] to routinely provide anorexic reduction over a wide range of database environments. However, a legitimate question remains as to whether the ideal choice of λ requires query and/or data-specific tuning. To assess this quantitatively, we show in Figure 9.7, the MSO_g values as a function of λ over the (0,100) percent range for a spectrum of query templates. The observation here is that the MSO_g values drop steeply with the use of covering enhancement and improve even further when λ is increased to 10%, and subsequently are relatively flat in the (10,30) percent interval, suggesting that our 20% choice for λ is a safe bet in general.

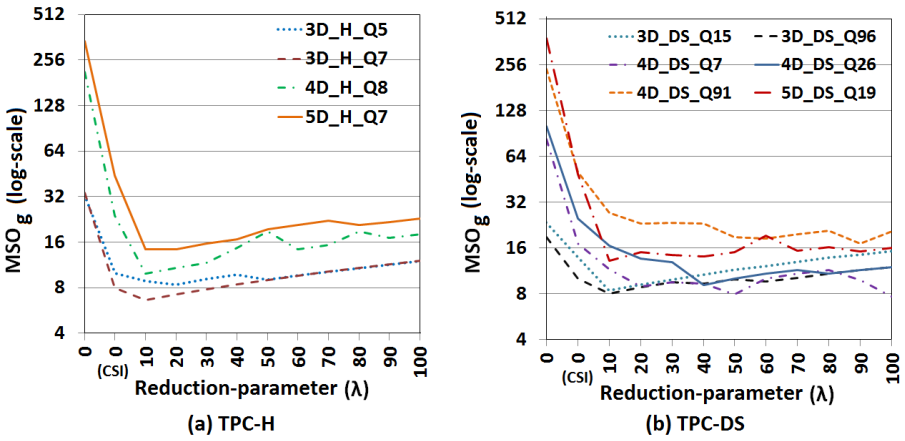


Figure 9.7: MSO_g vs Reduction-parameter (λ)

Chapter 10

MSO Bounds for Ad hoc Queries

As discussed in the previous chapters, the sub-optimality guarantees of the PB technique are contingent on the compile-time bouquet identification phase, whose overheads grow exponentially with the number of EPPs in the query. Such overheads are acceptable in canned query environments due to the expected amortization over repeated invocations for different instances of the template query. However, when we turn our attention to *ad hoc* queries, the preprocessing effort required in the query compile-time phase may not be practical.

In this chapter, we present a revamped algorithm $\mathsf{PB}_{\mathsf{AH}}$ to specifically address the provision of MSO guarantees for ad hoc queries. The differentiating factor compared to PB is the interaction style with the executor module, which is relaxed from the *black-box* mode of PB to a *white-box* engagement for $\mathsf{PB}_{\mathsf{AH}}$. To elaborate, at the end of each cost-budgeted execution, the query executor not only provides its completion status (black-box) but also provides the progress achieved by the execution in terms of the cardinality of the output which is then used to compute *lower bounds* on the error-prone selectivities. The white-box interaction helps in bounding the number of 1D executions required to cross an isosurface – specifically in terms of the number of EPPs in the query. Gratifyingly, over similar workloads to those considered in the previous chapters, $\mathsf{MSO}_{\mathsf{g}}$ values are typically *within a factor of 3* as compared to the

black-box bouquet technique.

Like PB, the $\mathsf{PB}_{\mathsf{AH}}$ technique can be implemented in a completely *non-invasive* manner wrt the database engine, if an API that supports output cardinality feedback at the end of each cost-budgeted execution is available. The infrastructure to support this facility already exists in almost every modern database systems in the form of cardinality tracking or monitoring modules. In fact, similar infrastructure is the basis of a recent feature in Micosoft SQL Server [5], termed as 'Live Query Statistics' [1].

To suit ad hoc queries, $\mathsf{PB}_{\mathsf{AH}}$ constructs the execution sequence in an *on-the-fly* manner, requiring only a few calls to the query optimizer before each cost-budgeted execution. For this purpose, $\mathsf{PB}_{\mathsf{AH}}$ incorporates two new modules: (1) *cost-budgeted planning*, to create a 1D execution having maximum selectivity learning potential for the given cost-budget and (2) *intermediate query injection*, to attack 1D subproblems for a query with multiple EPPs. Again, both these modules are straightforward to implement through API level interaction with the database engine, as explained later in this chapter.

We first show that for any query with single error-prone predicate, $\mathsf{PB}_{\mathsf{AH}}$ can construct an 'on-the-fly' execution sequence identical to that of PB and hence retains $\mathsf{MSO}_{\mathsf{g}}$ of 4. Next, we generalize the $\mathsf{PB}_{\mathsf{AH}}$ technique for the multi-dimensional version of the problem by using a decomposition approach, i.e., 'solve the multi-dimensional problem by attacking its constituent 1D subproblems'. With this approach, we show that the maximum number of executions to cross any isosurface is upper bounded by $\frac{D(D+1)}{2}$ where D is the number of error-prone selectivities, leading to MSO guarantees that are a lower order polynomial in D. Note here that $\mathsf{PB}_{\mathsf{AH}}$ does not require to enumerate any isosurface, hence it are referred to as *virtual* isosurface and the ESS is referred to as virtual ESS (vESS) during this chapter.

10.1 PB$_{AH}$ for 1D vESS

The PB$_{AH}$ technique for a 1D query constructs an execution sequence identical to that of PB, we first give a recap of the PB technique, followed by the difference in their construction mechanism.

Summary of PB in 1D vESS Recall from Section 4.1.1 that the PB algorithm for 1D vESS is a sequence of cost-budgeted executions $\{P_1, cost(IC_1)\}, \{P_2, cost(IC_2)\}, ..., \{P_m, cost(IC_m)\}$ where the cost budgets corresponding to IC_1, IC_2, ..., IC_m are geometrically distributed with initial cost value a and common ratio r such that $cost(IC_m) = C_{max}$ and $\frac{a}{r} < C_{min} \leq cost(IC_1) = a$. This sequence of cost-budgeted executions is constructed by first enumerating the optimal performance curve across the entire 1D selectivity range, and then identifying appropriate selectivity locations q_1, q_2, ..., q_m, on the optimal curve such that $P_{opt}(q_k)$=P_k and $c_{opt}(q_k)$= $cost(IC_k)$.

Here, the novel aspect of PB$_{AH}$ technique is that the generic execution IC_k is created without enumerating the entire selectivity range: Assuming $q_a > q_{k-1}$, we first identify the selectivity location q_k from the selectivity range (q_{k-1}, q_{max}], and then use it to create the cost-budgeted execution for IC_k. According to Lemma 4.1, this execution completes the query if $q_a \in (q_{k-1}, q_k$], in which case the remaining executions (for IC_{k+1} onwards) are not required to be identified. Otherwise if terminated, it implies that $q_a > q_k$ and the same process can be repeated to find q_{k+1}. The task of finding the next selectivity location q_k is fulfilled by a new module, *cost-budgeted planning* (CBP), described next.

10.1.1 Cost-Budgeted Planning (CBP)

Here, we find a query location with known optimal cost from among the locations in a 1D selectivity range and return the optimal plan choice for it. Let us denote the lower bound and upper bound on selectivity with q_{lb} and q_{ub}, having default values 0 and 1, respectively.

Specifically, the module takes the query Q and a cost value C_{pick} as input, along with selectivity bounds q_{lb} and q_{ub} such that $c_{opt}(q_{lb}) < C_{pick} \leq c_{opt}(q_{ub})$. The aim is to identify an

intermediate selectivity location, denoted with q_{pick}, such that $q_{lb} \prec q_{pick} \preceq q_{ub}$ and $c_{opt}(q_{pick}) = C_{pick}$, and then return the plan $P_{opt}(q_{pick})$. This process is illustrated in Figure 10.1 using the green curve to represent *virtual* optimal cost surface. As discussed in Chapter 3, the optimal cost curve is smooth and monotonic which implies that: (1) there exists *(exactly) one* location with optimal cost C_{pick} between q_{lb} and q_{ub}, (2) q_{pick} can be location using binary search. Hence, cost-budgeted planning is implemented as a binary search between q_{lb} and q_{ub} through repeated calls to the query optimizer module. Once q_{pick} is identified, we return the plan choice $P_{opt}(q_{pick})$ to create a cost-budgeted execution E_k with $\phi(E_k) = P_{opt}(q_{pick})$ and cost-budget $\omega(E_k) = C_{pick}$.

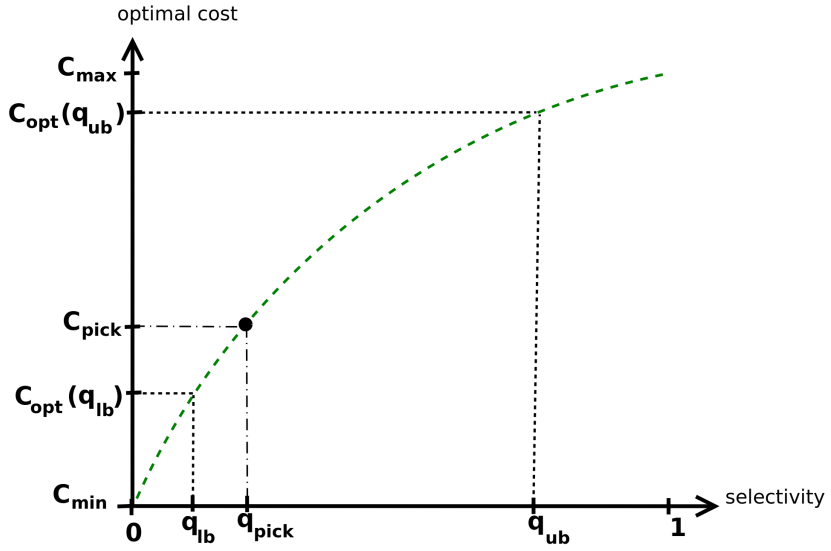


Figure 10.1: Cost-budgeted planning (CBP)

Alternative Implementation We can also use *exponential search* [13] whenever $q_{lb} \ll q_{ub}$ and $c_{opt}(q_{lb}) \ll c_{opt}(q_{ub})$, since our requirement of $C_{pick} = 2 \times c_{opt}(q_{lb})$ implies that q_{pick} is expected to be much closer to q_{lb} than q_{ub}.

Discussion The 1D algorithm for $\mathsf{PB}_{\mathsf{AH}}$ using CBP is outlined in Algorithm 4. Overall, since the execution sequence for any q_a is identical to that of PB, we conclude that $\mathsf{PB}_{\mathsf{AH}}$ retains $\mathsf{MSO}_{\mathsf{g}}$ of 4.

Algorithm 4: $\mathsf{PB}_{\mathsf{AH}}$ (1D)

$q_{lb} = 0$
$q_{ub} = 1$
// for each cost step IC_k
for $k = 1$ *to* m **do**
  // cost-budgeted planning: search q_k in the range (q_{lb},q_{ub}] using *cost(IC_k)* and return optimal plan at q_k
  P_k = CBP(Q, q_{lb}, q_{ub}, $cost(IC_k)$)
  start executing plan P_k
  // perform cost-budgeted execution
  while *run_cost*(P_k) $\leq$ $cost(IC_k)$ **do**
    execute P_k
    if P_k *completes execution* **then**
      return query result
    end
  end
  terminate P_k and discard partial results
  $q_{lb} = q_k + \epsilon$
end

10.1.2 Characteristic of 1D Executions

Selectivity Learning Potential (SLP) for a cost-budgeted execution For a query with 1 EPP, the SLP of a cost-budgeted execution $E = (P, C)$ is defined to be the *maximum* selectivity value that can be completely learned by the execution E and is denoted with q_C^P.

The SLP of an execution is given by the selectivity value at the intersection between the cost curve for the plan P and the cost step C. Figure 10.2 visually shows the SLP for executions created for cost-budgets $C1$ and $C2$ by using plans $P1$ and $P2$. Clearly, for cost-budgeted $C1$, execution with plan $P1$ has more SLP than with $P2$, while for cost-budget $C2$, more SLP corresponds to the execution that uses plan $P2$.

SLP of execution created using CBP Now, since CBP always chooses the plan from the intersection of cost-step with virual optimal cost curve, i.e. ,the *infimum* of cost-curves across all plans, the resulting execution has *maximum* SLP for the given cost-budget. As an implication,

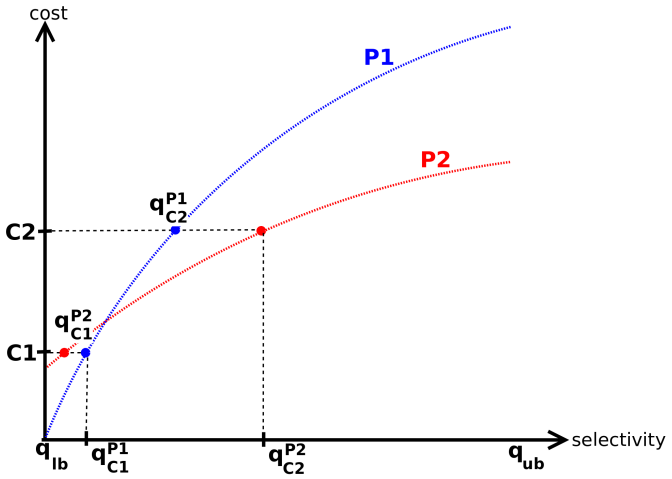


Figure 10.2: SLPs for $P1$ and $P2$ for costs $C1$ and $C2$

the executions performed by the 1D bouquet algorithm satisfies the following characteristics:

- when an execution with cost-budget C is terminated, it implies that optimal cost of solving the 1D problem is lower bounded by C,
- the only completed execution solves the 1D problem by learning the actual selectivity for q_a with at most twice the optimal cost required for solving the 1D problem.

10.2 $\mathsf{PB}_{\mathsf{AH}}$ for 2D vESS

The $\mathsf{PB}_{\mathsf{AH}}$ technique for 1D query was simply to find the plan on only the next cost step, rather than enumerating the entire optimal curve. A straightforward extension of this approach for a query with multiple EPPs would be to identify the next multi-dimensional isosurface and process plans present on it. However, since the plan-density reduction enhancements require to first enumerate the isosurface which is not feasible in ad hoc environment, this may lead to large number of executions for the isosurface and hence impractically large $\mathsf{MSO}_{\mathsf{g}}$ values. Instead, we take a *decomposition* approach where the multi-D problem is solved by attacking its 1D subproblems.

We first explain the technique using EQ2, a 2D query on TPC-H schema, as shown in Figure 10.6 which enumerates orders of cheap parts from suppliers with low account balances.

For this query, the two EPPs correspond to $P \bowtie L$ and $S \bowtie L$.

```
select *
from
    part P, lineitem L, supplier S, orders O
where
    P.p_partkey = L.l_partkey and S.s_suppkey = L.l_suppkey and
    O.o_orderkey = L.l_orderkey and
    P.p_retailprice < 1000 and
    S.s_acctbal < 95
```

Figure 10.3: Example TPC-H Query EQ2

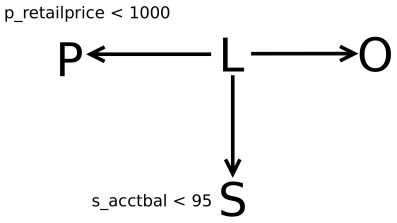


Figure 10.4: Join Graph for EQ2

10.2.1 Intermediate Queries

Given any SPJ query Q, it can be represented using a join-graph G with relations as vertices and join-predicates as edges. Also, the composite relation formed by combining relations in G corresponds to the final output relation of Q. For instance, Figure 10.4 shows the join graph for EQ2, where each directed edge represents a K-FK join predicate (pointing to K-side) and the composite relation $PSLO$ corresponds to the output relation of EQ2. Further, a *connected subgraph* of G corresponds to a potential *intermediate relation* in the execution plan for Q. The number of intermediate relations for a given query depends on the number of base relations and the structure of the join graph. For instance, a star query with n base relations has $(2^{n-1} - 2)$ intermediate relations e.g. the 6 intermediate relations for the star query EQ2 with 4 base relations are shown in Figure 10.5.

Now, for each intermediate relation S, we can construct an *intermediate query* by selecting from Q the relations, base-predicates and join-predicates that correspond to subgraph S and

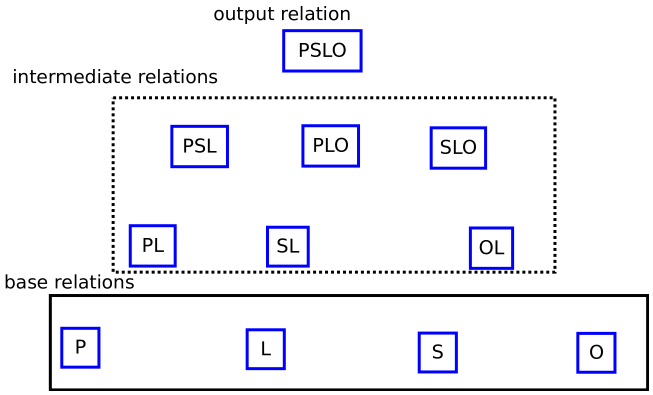


Figure 10.5: Intermediate relations for the example query EQ2

denote it with Q^S. For instance, query text corresponding to intermediate relation PLO, i.e. Q^{PLO} is shown below:

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

Figure 10.6: Intermediate query for PLO

10.2.2 Decomposition into Subproblems

For each intermediate relation (as well as output relation), the output cardinality can be written as a function of the base relation cardinalities and the join-predicate selectivities. For instance, denoting the selectivity of $P \bowtie L$ with x and that of $S \bowtie L$ with y, and knowing the selectivity for K-FK join $O \bowtie L$ to be $\frac{1}{|O|}$, the cardinality expressions for the intermediate relations corresponding to EQ2 are shown in Figure 10.7. While the current example has only *join* predicates only, the concepts are easily extendible to error-prone base predicates as well.

Now, the vESS dimensionality of any intermediate query (as well as the query itself) is given by the number of unknowns in its cardinality expression. Clearly, the dimensionality for these queries range from 0 to D and they can be partitioned on the basis of their unknowns.

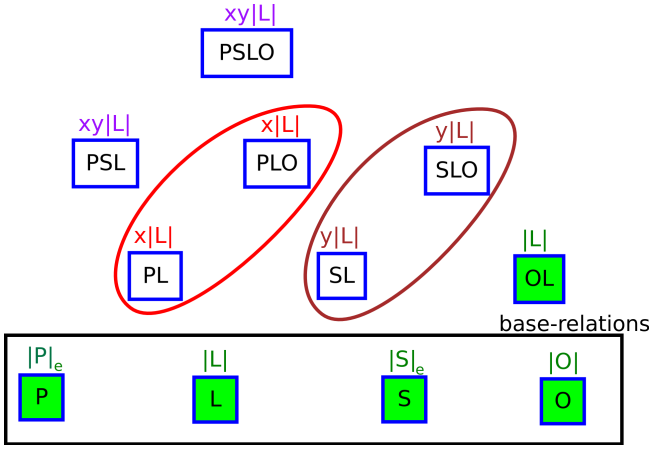


Figure 10.7: Intermediate relations and their cardinality expressions

The intermediate queries with known cardinalities are denoted with $\mathsf{Q}^{\{\phi\}}$ and are distinguished using box filled with green color in Figure 10.7. Further, 1-dimensional queries are highlighted with closed curves in Figure 10.7 and denoted with $\mathsf{Q}^{\{p\}}$ where p represents the selectivity dimension(s). Thus, we have:

- $\mathsf{Q}^{\phi} = \{\mathsf{Q}^{OL}\}$
- $\mathsf{Q}^{\{x\}} = \{\mathsf{Q}^{PL}, \mathsf{Q}^{PLO}\}$
- $\mathsf{Q}^{\{y\}} = \{\mathsf{Q}^{SL}, \mathsf{Q}^{SLO}\}$
- $\mathsf{Q}^{\{xy\}} = \{\mathsf{Q}^{PSL}, \mathsf{Q}^{PSLO}\}$

Here, both $\mathsf{Q}^{\{x\}}$ and $\mathsf{Q}^{\{y\}}$ are 1D subproblems for the 2D example query. Clearly, the number of 1-dimensional subproblems for a D-dimensional query is given by D, one corresponding to each error-prone selectivity. The subproblem $\mathsf{Q}^{\{xy\}}$ is a 2D problem which will be converted into 1D subproblem after one of the selectivity is completely learned.

10.2.3 Solving 1D Subproblems Using Repeated Invocations of CBP

The module CBP creates a sequence of executions with maximum SLP for a given 1D query. But, in the case of 1D subproblem of a higher-dimensional problem, it has been shown above

that each 1D subproblem may correspond to multiple intermediate queries, each of which has its own optimal cost curve.

Here, the execution for dimension x is identified by using repeated invocations of CBP once for each intermediate query in the 1D subproblem $\mathsf{Q}^{\{x\}}$, and choosing the one with maximum SLP. This idea is visually highlighted through Figure 10.8 where the red dot corresponds to larger selectivity as compared to the blue dot.

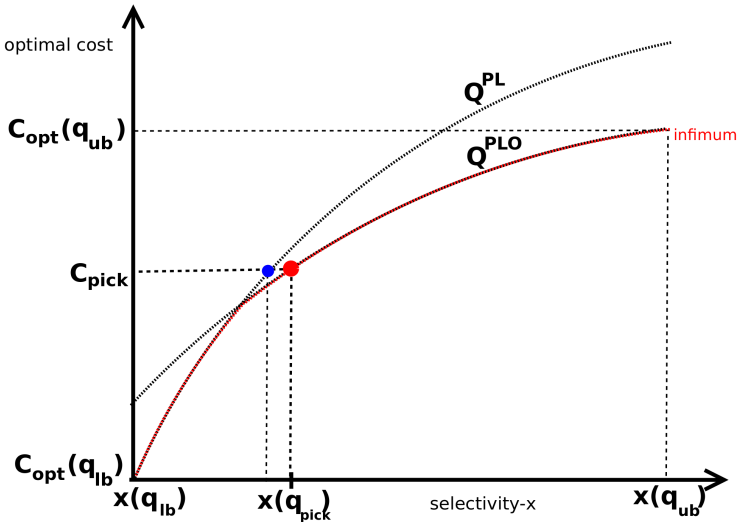


Figure 10.8: Execution with maximum SLP using repeated invocations of CBP

One execution per 1D subproblem Since the above execution corresponds to maximum learning potential for dimension x, any other execution created using a intermediate query in $\mathsf{Q}^{\{x\}}$ cannot improve upon the selectivity learning with the same cost-budget. Hence, generalized CBP ensures that only *one execution is sufficient* for a given 1D subproblem, i.e. combination of a cost-budget and a query set $\mathsf{Q}^{\{x\}}$.

10.2.4 Implications of 1D Executions

Terminated execution If the execution for $\mathsf{Q}^{\{x\}}$ with budget $cost(IC_k)$ is terminated, then it means that any plan for Q that uses intermediate relations from the set $\mathsf{Q}^{\{x\}}$ cannot complete the query within budget $cost(IC_k)$.

Completed execution If the execution for $\mathsf{Q}^{\{x\}}$ *completes* within its budget $cost(IC_k)$, then it implies that the actual selectivity x can be *completely learned* using the output cardinality feedback from the execution. For example, if the completed execution using Q^{PLO} returns 1000 tuples as output, then x can be computed by solving the corresponding cardinality equation $x|L| = 1000$. As a side-effect of such complete learning, the 2D subproblem $\mathsf{Q}^{\{xy\}}$ also gets converted to a 1D problem and leads to the following reorganization of the intermediate queries:

- $\mathsf{Q}^{\phi} = \{\mathsf{Q}^{OL}, \mathsf{Q}^{PL}, \mathsf{Q}^{PLO}\}$
- $\mathsf{Q}^{\{y\}} = \{\mathsf{Q}^{SL}, \mathsf{Q}^{SLO}, \mathsf{Q}^{PSL}, \mathsf{Q}^{PSLO}\}$

Intuitively, a terminated execution temporarily restricts the usage of intermediate relations for the current virtual isosurface and hence reduces the *space of plans* that can complete Q within budget $cost(IC_k)$. Note that all such dimensions are *revived* automatically once the budget is increased. On the other hand, a completed execution *permanently* reduces the variability in the cardinalities of the intermediate relations.

10.2.5 Number of 1D Executions to Cross an Isosurface

Only terminated executions If the execution for $\mathsf{Q}^{\{x\}}$ is terminated, we move on to create an execution for the other 1D subproblem $\mathsf{Q}^{\{y\}}$ with the same budget $cost(IC_k)$. If the second execution also gets terminated, then we can say that $c_{opt}(q_a) > cost(IC_k)$, since it is not possible to construct an execution plan for Q without using any intermediate node from $\mathsf{Q}^{\{x\}} \cup \mathsf{Q}^{\{y\}}$. As an instance from EQ2, the output relation $PSLO$ for Q cannot be constructed using only PSL and OL as intermediate relations. Overall, if all executions are terminated then there can be at most 2 executions with same budget $cost(IC_k)$, and it is also implied that the isosurface is crossed, i.e., $\text{cost}(q_a) > cost(IC_k)$.

With one completed execution In the other case when the execution for $\mathsf{Q}^{\{y\}}$ is completed, it leads to conversion of $\mathsf{Q}^{\{xy\}}$ to $\mathsf{Q}^{\{x\}}$, therefore reviving the x dimension for the current

isosurface. Since Q has now become a 1D problem, it requires only 1 execution per isosurface. Hence, there will be at most 3 executions with $cost(IC_k)$ as budget – e.g., (1) terminated for x, (2) completed for y, and (3) terminated/completed for x (converted from xy). If the last execution is completed then the 2D problem is solved, otherwise it continues as a 1D problem for the following isosurfaces.

10.3 $\mathsf{PB}_{\mathsf{AH}}$ for Multi-D vESS

In general, the algorithm for multi-D vESS is to identify the set of intermediate queries for each dimension and use them to find the execution with maximum SLP using repeated calls to the CBP routine. These 1D executions either result in complete selectivity learning of the dimension which result in reduced dimensionality of the vESS and also possible revival of other dimensions. On the other hand, in case we find that executions for all the remaining dimensions are *terminated*, we can increase the cost-budget and start performing executions for the next isosurface. The pseudocode for $\mathsf{PB}_{\mathsf{AH}}$ algorithm is shown in Figure 10.9.

10.3.1 Implications of 1D Executions

We first recall two crucial facts regarding our decomposition approach for multi-D vESS, i.e.,

1. Any 1D execution requires only one execution per cost-budget.
2. The number of 1D subproblems is exactly D.

Now, the importance of *completed* 1D executions is already evident from the fact that each such execution has the ability to reduce the dimensionality of the original problem one by one, finally reaching a 1D problem that requires only 1 execution per isosurface. Next, we present the generic result that establishes the ability of *terminated* executions to make progress in solving a D-dimensional problem.

Result 1 *For a given query* Q *with multiple error-prone selectivities, if the execution created using a 1D subproblem* $\mathsf{Q}^{\{x\}}$ *for budget* $cost(IC_k)$ *is terminated, then any plan for* Q *that uses*

PB$_{AH}$ **PseudoCode (multi-D)**

```
q_lb = (0,0, ..., 0);
q_ub = (1,1, ..., 1);                                   //initialize the selectivity bounds
D_e = {x_1,x_2,...,x_D}                                  //set of error-prone dimensions
k = 1;
complete = false;

while(complete == false) {                               //execution-loop until query completes
   Restart:
     for (x_i in D_e) {                                  //for each error-prone dimension
       Q^{x_i} = findIntermediateQueries(x_i);           //find set of 1D intermediate queries for the
required dimension x_i
       P^k_{x_i} = CBP(Q^{x_i}, x_i(q_lb), x_i(q_ub), cost(IC_k));   //find max SLP execution for x_i
through repeated calls to the CBP routine for different queries corresponding to x_i
       execInfo = CBE_WB(P^k_{x_i}, cost(IC_k));
       status = execInfo.status;
       selec_{x_i} = execInfo.learnedSelec;
       if(status == terminated)
           q_lb = update(q_lb, selec_{x_i});
       else {
           q_lb = update(q_lb, selec_{x_i});
           q_ub = update(q_ub, selec_{x_i});
           D_e = D_e - {x_i};                            //error-prone dimensions reduce by 1
           reorganize intermediate query partitions and find revived dimensions;
           goto Restart;
       }
     }

   if (D_e is non-empty)
     k++;                                                //jump to next isosurface
   else
     complete = true;
}
execute P_opt(q_ub);
```

Figure 10.9: Bouquet Algorithm for Adhoc queries

an intermediate relation from the set $\mathsf{Q}^{\{x\}}$ *cannot complete execution of* Q *within the budget* $cost(IC_k)$.

Proof: *Since the execution with maximum learning potential for* $\mathsf{Q}^{\{x\}}$ *and budget* $cost(IC_k)$ *is terminated, we know a lower bound on actual selectivity of* x. *The lower bound on* x *is such that all the intermediate relations in* $\mathsf{Q}^{\{x\}}$ *have cost more than* $cost(IC_k)$. *As a result, any plan for* Q *that uses one or more intermediate relations from* $\mathsf{Q}^{\{x\}}$ *costs more than* $cost(IC_k)$ *and hence cannot complete within budget* $cost(IC_k)$.

Unlike completed execution, a terminated execution does not have the permanent benefit of reducing the dimensionality and hence leads to wasted overheads in the bouquet execution sequence. But since any execution plan for Q must use at least one intermediate relation from the 1D subproblems of Q, the following result holds true:

Lemma 10.1 *For a query* Q *and budget cost*(IC_k)*, if the executions are terminated for all the 1D subproblems, then optimal cost for* Q *is certainly more than cost*(IC_k)*.*

Maximum Number of 1D executions to cross an isosurface Here, we prove results regarding the number of executions for a given cost-budget, using the above lemma:

Result 2 *For a query with D error-prone dimensions, the maximum number of terminated executions with a fixed cost-budget is D.*

Proof: *Since the initial number of 1D subproblems is D and if all executions are* terminated *then none of the subproblems is revived in the process.*

While the sequence of only terminated executions is bounded by D, the total number of 1D executions for a given virtual isosurface can still be more than D if one of the executions is completed. This is because a completed execution can cause reorganization of the intermediate queries across the subproblems and hence may lead to revival of the subproblems for which a terminated execution has already been performed.

Result 3 *For a query with D error-prone dimensions, the maximum number of executions (terminated or completed) with a fixed cost-budget is $\frac{D(D+1)}{2}$.*

Proof: *We know that a terminated execution cannot reduce the dimensionality of the problem and a completed execution can revive the subproblems for which execution has been terminated with the same cost-budget. Hence, to waste maximum execution effort before reducing the dimensionality due to a completed execution, there can be at most $(D-1)$ terminated 1D executions. After that, one 'completed' execution leaves a $(D-1)$ dimensional problem to be solved.*

Hence, the maximum number of 1D executions with a given cost-budget is given by the following recurrence relation:

$$NumExec(D) = [(D-1)+1] + NumExec(D-1) \quad with \quad NumExec(1) = 1$$

$$\Rightarrow NumExec(D) = D + (D-1) + (D-2) + ... + 2 + 1 = \frac{D(D+1)}{2}$$

10.3.2 $\mathsf{MSO_g}$ Analysis for Generic Multi-D vESS

With the above, we prove the following result regarding the $\mathsf{MSO_g}$ of the bouquet execution sequence.

Theorem 10.1 *The* $\mathsf{MSO_g}$ *is maximized when, for actual query location* $q_a \in (IC_{k-1},\ IC_k\ ]$, *the bouquet executions are distributed among the isosurfaces in the following fashion:* $[n_1,\ n_2,\ n_3,\ ...,\ n_{k-1},\ n_k] = [D,\ D,\ D,\ ...,\ D,\ \frac{D(D+1)}{2}]$. *With such an execution sequence,* $\mathsf{MSO_g}$ *is given by* $\left(\frac{r}{2}\right) D^2 + \left(\frac{r(r+1)}{2(r-1)}\right) D$.

Proof: *We prove the first part by contradiction and then compute the* $\mathsf{MSO_g}$ *for the given sequence.*

Assume that the cost-budget for the first isosurface that covers q_a *is* C. *The cost budgets for previously processed isosurfaces are:* $[\frac{C}{r^{k-1}},\ \frac{C}{r^{k-2}},\ \frac{C}{r^{k-3}},\ ...,\ \frac{C}{r},\ C]$.

First we compute the total cost for the proposed bouquet execution sequence and then show that any change to the above sequence will only decrease the total bouquet execution cost.

Step 1: *The total overheads for the proposed sequence are given by:*

$$c_B(*, q_a) = \left(D \times \frac{C}{r^{k-1}}\right) + \left(D \times \frac{C}{r^{k-2}}\right) + \left(D \times \frac{C}{r^{k-3}}\right) + ... + \left(D \times \frac{C}{r}\right) + \left(\frac{D(D+1)}{2} \times C\right)$$

$$= \left(D \times C \times (\frac{1}{r^{k-1}} + \frac{1}{r^{k-2}} + ... + \frac{1}{r})\right) + \left(\frac{D(D+1)}{2} \times C\right)$$

Step 2: *If only one of the selectivities is learned on the previous isosurface, it will cause reduction in the number of executions on the last isosurface and increase in executions on the previous surface, the new execution counts will be: [D, D, D, ..., $D + (D-1)$, $\frac{D(D-1)}{2}$] and the bouquet cost will be bounded by:*

$$c_B{}^{new}(*, q_a) \leq C \left(D \times (\frac{1}{r^{k-1}} + \frac{1}{r^{k-2}} + ... + \frac{1}{r}) + \frac{(D-1)}{r} + \frac{D(D-1)}{2} \right)$$

Now, the increase in cost is given by:

$$\Delta c_B(*, q_a) = C \left(\frac{(D-1)}{r} + \frac{D(D-1)}{2} - \frac{D(D+1)}{2} \right)$$

$$= C \left(\frac{(D-1)}{r} - D \right) = C \left(\frac{(D(1-r)-1)}{r} \right)$$

Hence, the bouquet cost can increase only if $r < 1 - \frac{1}{D}$ which is a contradiction since $r > 1$ and $D \geq 1$.

$\mathsf{MSO_g}$ ***computation:*** *Overall, this implies the total execution cost is upper bounded by $\frac{D(D+1)}{2}C + D \times (\frac{C}{r} + \frac{C}{r^2} + ...)$ while the oracle's cost for q_a is $\frac{C}{r} + \epsilon$ giving a MSO guarantee of:*

$$\mathsf{MSO_g} = r \times [\frac{D(D+1)}{2} + D \times (\frac{1}{r} + \frac{1}{r^2} + ...)]$$

$$\mathsf{MSO_g} = \frac{D(D+1)r}{2} + D \times (\frac{r}{r-1})$$

$$\mathsf{MSO_g} = \left(\frac{r}{2}\right) D^2 + \left(\frac{r(r+1)}{2(r-1)}\right) D$$

We get $\mathsf{MSO_g}$ to be $D(D+3)$ for $r = 2$. Further, the $\mathsf{MSO_g}$ expression reaches its minimum value for, $r = 1 + \sqrt{\frac{2}{D+1}}$, leading to $\mathsf{MSO_g} = \left(\sqrt{D} + \sqrt{\frac{D(D+1)}{2}} \right)^2$.

10.4 Performance Results

10.4.1 MSO bounds

Unlike PB where the MSO_g depends on the density of the plans on isosurfaces, PB_{AH} provides MSO_g as a monotonically increasing function of D, as highlighted in Figure 10.10.

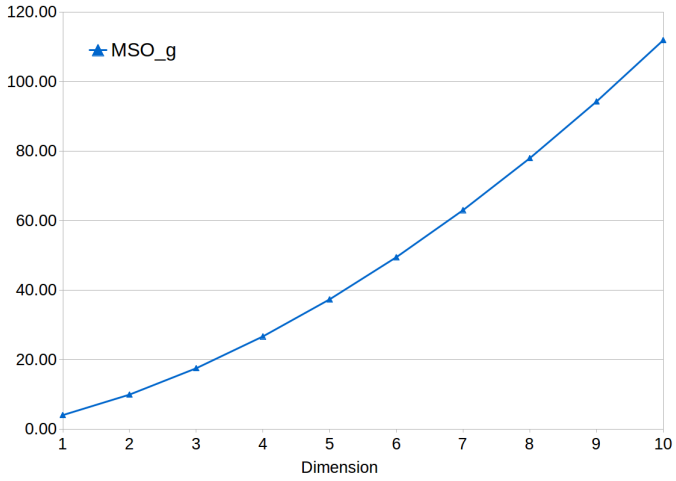


Figure 10.10: MSO_g increase with dimensions

Interestingly, even without any preprocessing effort, PB_{AH} could achieve MSO guarantees not much worse compared to PB– the comparison is given in Table 10.1. As we can observe, MSO_g does not degrade by more than a factor of 3 for most queries. Although there is no query in our suite for which PB_{AH} performing better than PB– it is, in principle, quite possible because PB_{AH} has the advantage of using feedback information from the executions and has a fixed number of executions per isosurface. On the other hand, PB depends on heuristic enhancements and off-line processing to reduce the isosurface plan-densities to practical levels.

10.4.2 Other empirical observations

In addition to the MSO bounds, we also performanced experiments to analyze PB_{AH} with regard to empirical MSO performance and number of plans used. It was found that, the empirical MSO

Query	$\mathsf{MSO_g}$ (Anorexic + CSI)	$\mathsf{MSO_g}$ (r=2) ($\mathsf{PB_{AH}}$)	$\mathsf{MSO_g}$ (optimal r) ($\mathsf{PB_{AH}}$)
3D_H_Q5	8.4	18	17.5
3D_H_Q7	7.2	18	17.5
3D_DS_Q15	9.2	18	17.5
3D_DS_Q96	8.8	18	17.5
4D_H_Q8	15	28	26.7
4D_DS_Q7	9.1	28	26.7
4D_DS_Q26	8.1	28	26.7
4D_DS_Q91	16	28	26.7
5D_H_Q7	15	40	37.3
5D_DS_Q19	15	40	37.3

Table 10.1: Guarantees in ad hoc query environment

performance was much better compared to the MSO bounds, this happened because the bounds take care of the worst case where the dimensions are *repeatedly revived* on many consecutive isocost surfaces – which rarely happened in practice. Overall, empirical MSO varied between 10 to 20, which was significantly better than corresponding values of MSO bounds specifically for high dimensional queries.

Also it was found that, for different q_a instances of same query, $\mathsf{PB_{AH}}$ uses different set of plans in the execution sequences – this is in contrast with PB which uses a small set of preidentified executions to execute any q_a in the ESS. This is expected because PB spends preprocessing effort to find a small execution sequence for the entire ESS using various enhancements like execution swallowing and execution covering, while $\mathsf{PB_{AH}}$ cannot utilize any enhancement and prepares the execution sequence for a given q_a in an on-the-fly manner. More importantly, $\mathsf{PB_{AH}}$ still provides empirical sub-optimality performance comparable to that of PB.

Chapter 11

Discussion

Having presented the mechanics and performance of the bouquet approach, we now take a step back and relook at the ways to relax some of the optional assumption made during the thesis. Next, we critique the bouquet technique, and then explore the deployment scenarios.

11.1 Revisiting Assumptions

In this section, we revisit certain issues that are very important in terms of their impact execution plan sub-optimality but were initially avoided using simplying assumptions. We made those assumptions so that the discussion can be focused on the primary issue of selectivity estimation errors. Now, we revisit these *optional assumptions* of the bouquet approach and propose possible direction(s) in which they can be addressed within the bouquet framework.

11.1.1 Selectivity Independence Assumption

Intra-relational predicate combinations

Usually, selectivity independence is employed to compute the overall selectitivity for a combination of intra-relational predicates – this is *not* necessarily true for the bouquet approach. The reason for the above property is that, due to the exploratory nature of plan bouquet ex-

ecution(s), the actual combined selectivity can be usually *discovered* by evaluating all filter predicates over the output received from the relational scan operator without any significant cost overhead. Of course, it is valid only when the predicate evaluation costs are insignificant compared to scan operator's cost.

For example, consider the following query over standard TPC-H schema.

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

This query gives the order details of *cheap* and *small* parts in the data. Here, the predicate l_orderkey = o_orderkey is join-predicate among the relations lineitem and orders and l_partkey = p_partkey is a join-predicate among the relations lineitem and parts – the remaining two predicates are filter predicates on the relation part.

The standard query processing architecture employs binary operators for joins, i.e., joins two relations at a time. Due to this fact, an operator can employ only one of the join-predicate, but this restriction is not there on scan operators for base relations. Hence, combined selectivity for filter predicates p_retailprice < 1000 and p_type = 'SMALL' need not be estimated but can be discovered using a single scan operator.

Independence among join-predicates

Until now in the thesis, selectivity independence assumption has been employed whenever it is required to compute the combined selectivity of more than one join-predicate. Next, we propose a possible direction in which bouquet approach can be extended without making the independence assumption among the join-predicates.

Specifically, join-predicate independence assumption implies that the selectivity of a predicate join-predicate does not depend on whether it has been used before or after employing another

join-predicate. Consider the cardinality expressions of query EQ2 (in Chapter 10) repeated in Figure 11.1a. Here, the selectivity of $(P \bowtie L)$ join remains x even when it is used in $P \bowtie (S \bowtie L)$, where it is employed after $S \bowtie L$ and results in cardinality expression $xy|L|$.

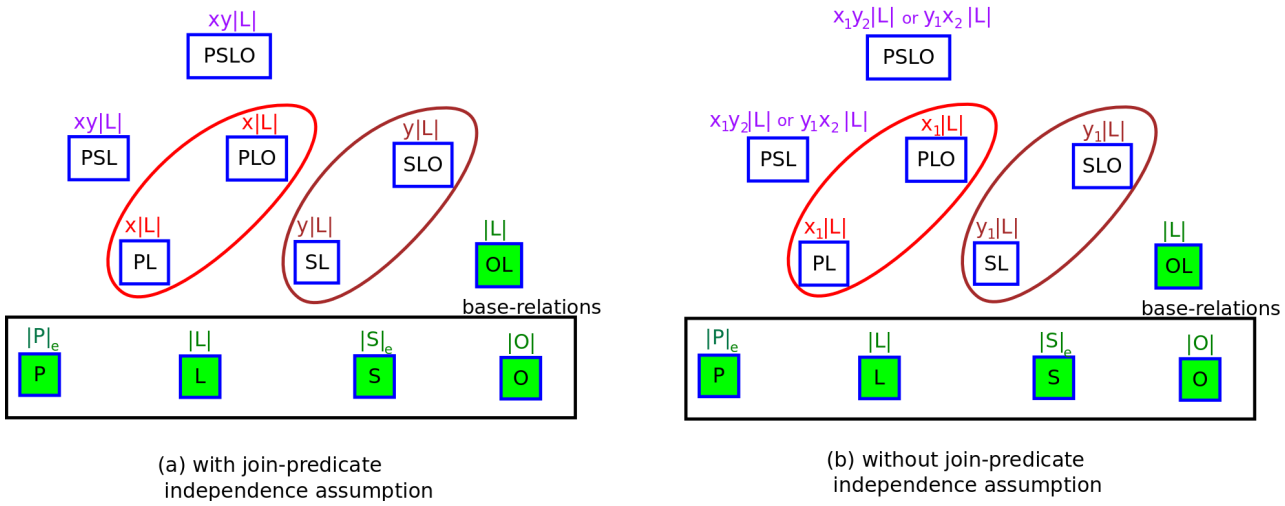


Figure 11.1: Cardinality expressions with and without join-predicate independence assumption

Consider the case when the join-predicates do not have independent selectivities, let us denote the selectivity of join-predicate $P \bowtie L$ with x_1 when used directly and x_2 when used after join-predicate $S \bowtie L$. Similarly, denote the selectivity of $S \bowtie L$ with y_2 when used after $P \bowtie L$ and y_1 when used directly. Note that, these selectivities are still independent wrt the predicate $O \bowtie L$ since every tuple will satisfy the join-predicate with relation O as it does not face any loss of keys. The resulting cardinality expressions are shown in Figure 11.1b. Here, the cardinality of PSL is $y_1x_2|L|$ when the join-order is $P \bowtie (S \bowtie L)$ and $x_1y_2|L|$ when the join-order is $(P \bowtie L) \bowtie S$ – clearly $y_1x_2|L| = x_1y_2|L|$.

With independence assumption (**INDEP** case), it was a 2D problem where the cardinality of all the intermediate relations (as well as the output relation) could be computed after learning the actual value of x and y. While in the absence of independence assumption (**DEP** case), it is a 3D problem as it need to learn the actual values of x_1, y_1, y_2 or x_1, y_1, x_2 to compute the cardinalities for all the relations. To elaborate, while the completed execution for x_1 converts

the 2D subproblem $\{PSL, PSLO\}$ from x_1y_2 into 1D subproblem with dimension y_2, it does not merge with the dimension y_1 for $\{SL, SLO\}$.

The new problem is equivalent to a problem with the following variables: $X = x_1|L|$, $Y = y_1|L|$ and $Z = x_1y_2|L| = y_1x_2|L|$, where X, Y, Z are the output cardinalities of $\{PL, PLO\}$, $\{SL, SLO\}$ and $\{PSL, PSLO\}$, respectively. Once X, Y and Z are known, all the variable x_1,x_2, y_1, and y_2 can be computed from them. Thus, in general, the dimensionality without independence assumption is the number of intermediate relations with unique cardinality expressions, instead of the number of error-prone predicates, and can be denoted with D^*. Further, since usage of CBP ensures one execution (completed or terminated) for each of these variables, the maximum number of executions per isosurface is given by D^*. To put things in perspective, the new dimensionality D^* for a star-query with all D join predicates being error-prone will be given by $2^D - 1$. Coincidentally for the example query EQ2, the adverse impact is not significant and $\mathsf{MSO_g}$ increases from 10 to 12 (=4 × 3) when join-predicate independence assumption is relaxed. Similarly, the increased values of $\mathsf{MSO_g}$ for our test suite of queries is given in Table 11.1. It is an interesting future work to analyze the tightness of these $\mathsf{MSO_g}$ values.

Query	$\mathsf{MSO_g}$ (r=2) **(INDEP)**	$\mathsf{MSO_g}$ **(DEP)**	**Query Graph** (# relations)
3D_H_Q5	18	24	chain(6)
3D_H_Q7	18	24	chain(6)
3D_DS_Q15	18	24	chain(4)
3D_DS_Q96	18	28	star(4)
4D_H_Q8	28	48	branch(8)
4D_DS_Q7	28	60	star(5)
4D_DS_Q26	28	60	star(5)
4D_DS_Q91	28	88	branch(7)
5D_H_Q7	40	60	chain(6)
5D_DS_Q19	40	96	branch(6)

Table 11.1: MSO guarantees without join-predicate independence assumption

11.1.2 Perfect Cost Model

Thus far, we had catered to arbitrary errors in selectivity estimation, but assumed that the cost model itself was perfect. In practice, this is certainly not the case, but if the modeling errors were to be unbounded, it appears hard to ensure robustness since, in principle, the estimated cost of any plan could be arbitrarily different to the actual cost encountered at run-time. However, we could think of an intermediate situation wherein the modeling errors are non-zero but *bounded* – specifically, the estimated cost of any plan, given correct selectivity inputs, is known to be within a δ error factor of the actual cost. That is, $\frac{c_{estimated}}{c_{actual}} \in [\frac{1}{(1+\delta)}, (1+\delta)]$.

Our construction is lent credence to by the recent work of [87], wherein static cost model tuning was explored in the context of PostgreSQL – they were able to achieve an average δ value of around 0.4 for the TPC-H suite of queries. This "unbounded estimation errors, bounded modeling errors" framework is amenable to robustness analysis and leads to following result:

Theorem 11.1 *If the cost-modeling errors are limited to error-factor δ with regard to the actual cost, the bouquet algorithm ensures that:* $MSO_g = (1+\delta)^2 \rho \frac{r^2}{r-1}$

The effectiveness of this result is clear from the fact that, when $\delta = 0.4$, corresponding to the average in [87], the $\mathsf{MSO_g}$ increases by at most a factor of 2. Such low value of δ is also corroborated by the views of industry experts [62] based on their experience in real world scenarios.

$$C_{bouquet}(q_a) = \delta(a + ar + ar^2 + ... + ar^{k-1}) - ar^{k-2}(\delta - \frac{1}{\delta}) \tag{11.1}$$

And the corresponding cost for "oracle" algorithm is $\frac{ar^{k-2}}{\delta}$, causing

$$SubOpt(*, q_a) \leq \frac{\delta \frac{a(r^k-1)}{r-1} - ar^{k-2}(\delta - \frac{1}{\delta})}{\frac{ar^{k-2}}{\delta}} \tag{11.2}$$

$$MSO_{bounded\_modeling\_error} \leq \delta^2 MSO_{perfect\_model} - (\delta^2 - 1) \tag{11.3}$$

$$MSO_{bounded\_modeling\_error} \leq \delta^2 (MSO_{perfect\_model} - 1) + 1 \tag{11.4}$$

11.1.3 No known Selectivity Bounds and Lack of Absolute Metric

Until now, we have presented performance of the bouquet algorithm only on the sub-optimality based metrics, and also assumed that no bounds on selectivities are known. Here, we propose extensions to the bouquet approach where we consider the usage of selectivity bounds to improve execution performance and also propose another *absolute* performance metric for bouquet style execution approaches.

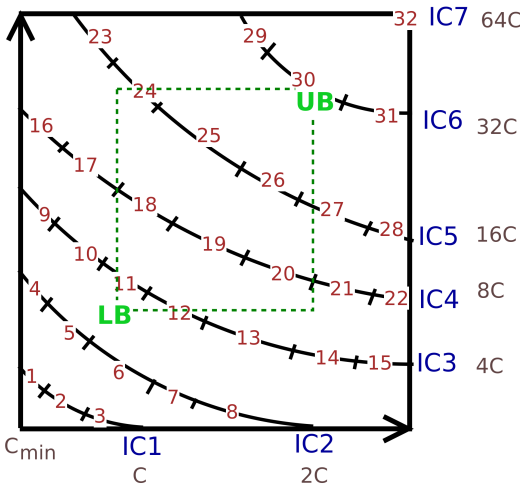


Figure 11.2: Utilizing lower bound (LB) and upper bound (LB) on selectivities

Using Selectivity Bounds While worst case guarantees in the absence of selectivity bounds are the primary benefit of the bouquet approach, the technique can also utilize lower and upper bounds on error-prone selectivities, if they are available. Specifically, these can be utilized to restrict and identify the relevant portion of the ESS which may help in improving the MSO performance by skipping irrelevant executions from the original cost-budgeted sequence, as

shown in Figure 11.2. In this Figure, the selectivity bounds help to skip 22 out of 32 executions and the execution sequence is only $\{E_{11}, E_{12}, E_{18}, E_{19}, E_{20}, E_{24}, E_{25}, E_{26}, E_{27}, E_{30}\}$.

An Absolute Metric Here, we analyze the total (absolute) execution cost for the bouquet and show that it is a function of the maximum isosurface plan-density along with the optimal execution cost of the terminal execution, i.e., execution corresponding to the upper bound on selectivities. Further, we observe that this absolute metric, i.e., total bouquet execution cost, can be improved by trading off with the MSO guarantee provided by the sequence.

Specifically, for 1D ESS with cost ratio r, we know that the worst case sub-optimality is bounded by $\text{MSO}_\text{g} = \frac{r^2}{r-1}$ and the total bouquet execution cost is given by $TAC = (1 + \frac{1}{r-1})C_{max}$. Figure 11.3 shows the variation of MSO_g and $\frac{TAC}{C_{max}}$ with increasing cost-ratio r. Both these metric decrease steeply in the range $(1, 2]$ after which MSO_g starts increasing while $\frac{TAC}{C_{max}}$ continues to decrease to asymptotically reach 1. For $r = 2$, TAC is $2C_{max}$ and when r is increased to 4, TAC decreases to $1.33C_{max}$ while MSO_g increases to 5.33.

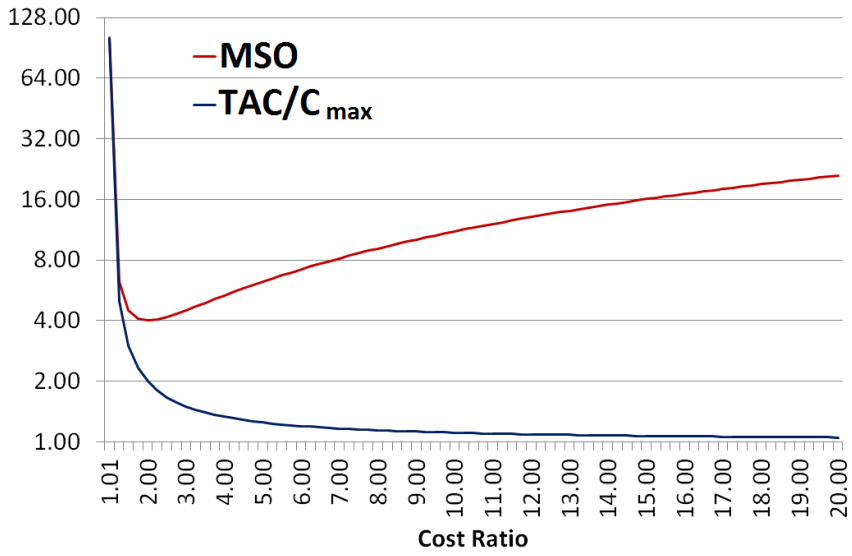


Figure 11.3: Variation of MSO_g and $\frac{TAC}{C_{max}}$ with cost-ratio r

11.2 Critique of Bouquet Approach

Generic limitations of plan-switching approaches Being a *plan-switching* approach, the bouquet technique suffers from the drawbacks generic to such approaches: Firstly, they are poor at serving *latency-sensitive* applications as they have to perforce wait for the final plan execution to return result tuples. Secondly, they are not recommended for update queries since maintaining transactional consistency with multiple executions may incur significant overheads to rollback the effects of the aborted partial executions. Finally, with single-plan optimizers, DBAs use their domain knowledge to fine-tune the plan using "plan-hints". But this is not straightforward in *plan-switching* techniques since the actual plan sequence is determined only at run-time. Notwithstanding the limitations, such techniques are now featured even in commercial products (e.g. [2]).

There are also a few problems that are *specific* to the bouquet approach, the first one is applicable to both PB and $\mathsf{PB}_{\mathsf{AH}}$ while the other two are specific to PB technique:

Unsuitable for small estimation errors The bouquet approach is intended for use in difficult estimation environments – that is, in database setups where accurate selectivity estimation is hard to achieve. However, when estimation errors are apriori known to be small (but selectivity bounds are not explicitly available), re-optimization techniques such as [65, 12], which use the optimizer's estimate as the initial seed, are likely to converge much quicker than the bouquet algorithm, which requires starting at the origin to ensure the first quadrant invariant. But, if the estimates were apriori guaranteed to be *under-estimates*, then the bouquet algorithm can also leverage the initial seed.

Changes in data-distribution While it is inherently robust to changes in data *distribution*, since these changes only shift the location of q_a in the existing ESS, the same is not true with regard to database *scale-up*. That is, if the database size increases significantly, then the original ESS no longer covers the entire error space. An obvious solution to handle this problem

is to recompute the bouquet from scratch, but most of the processing may turn out to be redundant. Therefore, developing incremental bouquet maintenance strategies is an interesting future research challenge.

Dimensionality of selectivity error space The dimensionality of error space can be huge for complex queries which has direct impact on the bouquet identification overheads as well as the MSO_g. In addition to error-prone selectivity dimensions, even parameterized predicates (if any) need to be included in the ESS causing further increase in bouquet identification overheads – although bouquet execution considers only appropriate subspace using the instantiated parameters at run-time. But, it is important to note that, a complex query does not necessarily imply a commensurately large number of error dimensions because: (i) The selectivities of base relation predicates of the form "*column op constant*" can be estimated accurately with current techniques; (ii) The join-selectivities for PK-FK joins can be estimated accurately if the entire PK-relation participates in the join. Still, techniques need to be developed in future to identify and remove those dimensions whose impact on MSO_g is negligible compared to others.

11.2.1 Deployments Aspects

Given the above discussion, the bouquet approach is currently recommended specifically for providing response-time robustness in large archival read-only databases supporting complex decision-support applications that are likely to suffer significant estimation errors. We expect that many of today's OLAP installations may fall into this category. We wish to highlight that from a deployment perspective, the bouquet technique is intended to *complementarily co-exist* with the classical optimizer setup, leaving it to the user or DBA to make the choice of which system to use for a specific query instance.

Chapter 12

Conclusions and Future Directions

Selectivity estimation errors resulting in poor query processing performance are part of the database folklore. While there has been many innovative research proposals that are capable of ameliorating the adverse impact of these errors on the execution performance, none of them provide any performance guarantees. The contribution of this thesis is a new query processing framework which is based on discovery of error-prone selectivities in a controlled manner and allows theoretical analysis of the performance as compared to an oracular system that magically knows the correct values for these selectivities.

Next, we summarize the conclusions of this thesis in the direction of achieving execution performance that is robust to selectivity estimation errors followed by outlining a number of potential areas for further research to achieve the larger goal of robustness in query processing.

12.1 Conclusions

In this thesis, we investigated a new approach to this classical problem, wherein the estimation process was completely discarded for error-prone predicates. Instead, such selectivities were progressively discovered at run-time through a carefully graded sequence of cost-budgeted executions from a "plan bouquet". The execution sequence, which followed a cost-doubling geometric progression, ensured that the overheads are bounded, thereby ensuring $\mathsf{MSO_g}$ of 4

times the plan cardinality of the densest isosurface. Also, incorporating randomized strategies in the above algorithm brought down the multiplicative factor of 4 to only 1.8 as the guarantee on the expected performance. To the best of our knowledge, such bounds have not been previously presented in the database literature.

We also proposed an efficient isosurface identification algorithm for pragmatic overheads during bouquet identification, and two compile time enhancements that significantly improved the worst case guarantees. Together they ensured that MSO_g was less than 20 across all the queries in our evaluation set, an enormous improvement compared to the MSO performance of the native optimizer, wherein this metric ranged from thousands to millions. Further, the bouquet's ASO performance was always either comparable to or much better than the native optimizer, with most of the query locations having a sub-optimality of less than 4. While the bouquet algorithm did occasionally perform worse than the native optimizer for specific query locations, such situations occurred at less than 1% of the locations, and the performance degradation was relatively small, a factor of 2 or less.

Finally, we also extended the bouquet approach to handle the query scenarios where the preprocessing effort at compile-time may not be feasible, including ad hoc queries and queries with larger number of error-prone dimensions. In this direction, we designed a revamped algorithm that exploits the output cardinality of the cost-budgeted executions to learn lower bounds on error-prone selectivities and generate the execution sequence in an on-the-fly manner. Even with its dynamic nature it follows a bound on the number of executions per virtual isosurface leading to performance guarantees as a function of number of error-prone dimensions. Interestingly, the MSO guarantees with this approach are found to be within a small factor of the values that could be achieved with the preprocessed bouquet sequence.

12.1.1 Relevance in Non-relational Systems

In recent times, the database community has seen a large number of attempts to suit different kinds of data processing requirements and do not resemble the traditional relational database

systems, which is the primary focus of this thesis. Still, we wish to highlight that the proposed techniques are useful in any system that uses selectivity inputs to choose the ideal execution plan and the chosen plans satisfy the monotonicity and smoothness assumption. In general, the underlying ideas have the potential to improve query processing performance in other systems as well, due to below mentioned reasons.

The first and foremost advantage with bouquet technique is that it can achieve reasonable query processing performance even in the complete *absence* of any statistical information about the data – the mandatory information required is only the base-relation cardinalities which can be trivially managed. In contrast, the statistical information seems vital for selection of reasonable plans in any system where there are large number of executions choices having widely varying execution times. Many data processing systems handle this absence of meta-data information by either directly using imperative code or commodity hardware to execute otherwise highly suboptimal plans, which is not always sufficient to achieve near-optimal performance.

Further, the decision to start exploration with a plan that is optimal for very small selectivity and to maintain sufficient cost gap (e.g. geometrically increasing) between the exploratory executions ensures that the relative overheads increase quite slowly as compared to the absolute overheads. These ideas are generic and can be used even if the system does not really have a cost-based optimizer and/or satisfies the optimal cost monotonicity properties in piecewise manner.

Closing Statement In closing, the bouquet approach promises an easy to deploy solution with guaranteed performance and repeatability in query execution, features that had hitherto not been available, thereby opening up new possibilities for robust query processing.

12.2 Future Directions

This thesis gives an initial set of query processing techniques that are easy to deploy and amenable to theoretical performance analysis. We now turn our attention to some interesting

future directions which highlight the fertile research ground provided by the plan bouquet framework:

1. **Dynamically learned cost model:** While we have analyzed the impact of the cost modeling errors on the performance guarantees in terms of a known factor δ, it is interesting to see whether the unpredictable run-time conditions due to fluctuating workload situations can also be handled in the learning-based framework of plan bouquet.

2. **Bouquet technique for a workload:** In this thesis, our focus was to analyze the execution performance overheads for a single independent query in the system. With more queries in the simultaneous workload there are more opportunities in sharing the selectivity learning and even sharing the execution subplans, which hints towards a potential area for further research.

3. **Removing the join-predicate independence assumption:** In this direction, we have already provided a first cut solution by extending the 'on-the-fly' bouquet construction algorithm. But this direction still needs further research to explore the looseness of these MSO bounds.

4. **Tightness of performance guarantees:** In this thesis, we presented suboptimality guarantees for multi-dimensional queries and also proposed enhancements that empirically improve the guarantees to a significant level. But it was not analyzed whether the proposed upper bounds are tight or there is still scope of further improvement – a theoretical analysis in this regard is required to fully understand the capabilities of discovery based query processing techniques.

5. **Exploiting the behavior of optimal cost profile:** In this work, the only requirement is that the optimal cost profile should be monotonic wrt increase in selectivity inputs. It has also been observed that, the slope of the optimal cost profile actually decreases as we

move along the selectivity dimensions. This *concave* behavior of optimal cost profile helps indirectly by improving the effectiveness of the execution covering enhancement but it has not been included in the formal analysis, which may possibly lead to improved guarantees for at least those queries where concave behavior can be inferred from the query structure – this is another interesting direction for future research.

6. **Reduction in preprocessing overheads:** The NEXUS algorithm brings the preprocessing requirement down by at least an order of magnitude for most high-dimensional queries, but most of the plan executions identified in this process are later skipped by the plan swallowing and covering enhancements. Thus, it can be interesting to see whether it is possible to directly identify or at least approximate the execution sequence (after both enhancements) without actually identifying the much larger initial sequence through NEXUS.

7. **Ranking of error-prone dimensions:** It is clear, even with NEXUS algorithm, the overheads of identifying complete bouquet sequence still increase at a significantly high rate when the dimensionality of selectivity error space increases. One possible direction to keep the overheads in control is to rank the error-prone dimensions and skip the low-rank dimensions such that there is a tradeoff between the bouquet identification overheads and the degradation in MSO guarantee due to the left over dimension(s).

8. **Incremental bouquet maintenance:** The bouquet algorithm for canned queries work by identifying a fixed execution sequence at compile-time and then using the deterministic sequence to handle any query instance in the selectivity error space. While this bouquet sequence can withstand any change in the distribution of data, any change in *data scale* makes it unsuitable for query instances in the new selectivity error space. To handle this, a straightforward approach is to compute the entire execution sequence from scratch, but this may cause redundant work in the exploration phase. Hence, it would be interesting

to see whether the additional work required to update the bouquet execution sequence can be minimized.

Bibliography

[1] Live query statistics. https://msdn.microsoft.com/en-IN/library/dn831878.aspx, October 2015. 94

[2] Optimizer with oracle database 12c. www.oracle.com/ocom/groups/public/@otn/documents/webcontent/1963236.pdf, 2015. 118

[3] Postgresql 8.4. www.postgresql.org/docs/8.4/static/release.html, 2015. 85

[4] Using a selectivity clause to influence the optimizer. www.ibm.com/developerworks/data/library/tips/dm-0312yip/, 2015. 77

[5] Using the use plan query hint. technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx, 2015. 77, 94

[6] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. Rox: The robustness of a run-time xquery optimizer against correlated data. In *Proc. of the 26th Intl. Conf. on Data Engg.*, ICDE '10, pages 1185–1188, 2010. 21

[7] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '99, pages 181–192, 1999. 7, 19, 22

[8] G. Antoshenkov. Dynamic query optimization in rdb/vms. In *Proc. of the 9th Intl. Conf. on Data Engg.*, ICDE '93, pages 538–547, 1993. 21

[9] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '00, pages 261–272, 2000. 21, 25, 27

[10] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '05, pages 119–130, 2005. 7, 12, 21, 22, 26

[11] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005. 25

[12] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '05, pages 107–118, 2005. 7, 21, 22, 24, 25, 26, 86, 118

[13] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976. 96

[14] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, VLDB '05, pages 757–768, 2005. 21

[15] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive parametric query optimization. *IEEE Trans. on Knowl. and Data Eng.*, 21(4):582–594, 2009. 26, 32

[16] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth Scan: Statistics-Oblivious Access Paths. In *Proc. of 31st Intl. Conf. on Data Engg.*, ICDE '15, 2015. 21, 22, 27

[17] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '02, pages 263–274, 2002. 20

[18] Nicolas Bruno and Surajit Chaudhuri. Conditional selectivity for statistics on query expressions. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '04, pages 311–322, 2004. 20

[19] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: A multidimensional workload-aware histogram. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '01, pages 211–222, 2001. 19

[20] Surajit Chaudhuri. Query optimizers: Time to rethink the contract? In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '09, pages 961–968, 2009. 18

[21] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. Variance aware optimization of parameterized queries. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '10, pages 531–542, 2010. 7, 21, 22, 26, 32

[22] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '99, pages 263–274, 1999. 20

[23] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. of 1998 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '98, pages 436–447, 1998. 20

[24] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. A pay-as-you-go framework for query execution feedback. *Proc. of VLDB Endow.*, 1(1):1141–1152, 2008. 20, 26

[25] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. Exact cardinality query optimization for optimizer testing. *Proc. of VLDB Endow.*, 2(1):994–1005, August 2009. 4, 18

[26] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '94, pages 161–172, 1994. 20

[27] Stavros Christodoulakis. *Estimating Selectivities in Data Bases*. PhD thesis, Toronto, Ont., Canada, Canada, 1982. AAI0537828. 19

[28] Stavros Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Syst.*, 9(2):163–186, 1984. 19

[29] Francis Chu, Joseph Halpern, and Johannes Gehrke. Least expected cost query optimization: What can we expect? In *Proc. of the 21st ACM Symposium on Principles of Database Systems*, PODS '02, pages 293–302, 2002. 7, 21, 22, 26

[30] Francis C. Chu, Joseph Y. Halpern, and Praveen Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. of the 18th ACM Symposium on Principles of Database Systems*, PODS '99, pages 138–147, 1999. 21

[31] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. 1, 83

[32] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '94, pages 150–160, 1994. 21, 26

[33] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '01, pages 199–210, 2001. 19

[34] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in databases*, 1(1):1–140, 2007. 25

[35] A. El-Helw, I.F. Ilyas, Wing Lau, V. Markl, and C. Zuzarte. Collecting and maintaining just-in-time statistics. In *Proc. of the 23rd Intl. Conf. on Data Engg.*, ICDE '07, pages 516–525, 2007. 20

[36] Cristian Estan and Jeffrey F. Naughton. End-biased samples for join cardinality estimation. In *Proc. of the 22nd Intl. Conf. on Data Engg.*, ICDE '06, pages 20–, 2006. 20

[37] FedorV. Fomin and AlexeyA. Stepanov. Counting minimum weighted dominating sets. In *Computing and Combinatorics*, volume 4598 of *Lecture Notes in Computer Science*, pages 165–175. 2007. 71

[38] Campbell Fraser, Leo Giakoumakis, Vikas Hamine, and Katherine F. Moore-Smith. Testing cardinality estimation models in sql server. In *Proc. of the 5th Intl. Workshop on Testing Database Systems*, DBTest '12, pages 1–7, 2012. 18

[39] Minos Garofalakis and Phillip B. Gibbons. Wavelet synopses with error guarantees. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '02, pages 476–487, 2002. 19

[40] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '01, pages 461–472, 2001. 19

[41] Goetz Graefe. Robust query processing. In *Proc. of the 27th Intl. Conf. on Data Engg.*, ICDE '11, page 1361, 2011. 18

[42] Goetz Graefe. New algorithms for join and grouping operations. *Comput. Sci.*, 27(1):3–27, February 2012. 21, 22, 27

[43] Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. Robust query processing (dagstuhl seminar 12321). http://www.dagstuhl.de/12321, 2012. 18

[44] Goetz Graefe, Arnd Christian König, Harumi Anne Kuno, Volker Markl, and Kai-Uwe Sattler. Robust query processing (dagstuhl seminar 10381). http://www.dagstuhl.de/10381, 2010. 17, 18, 22, 29

[45] Peter J. Haas, Ihab F. Ilyas, Guy M. Lohman, and Volker Markl. Discovering and exploiting statistical properties for query optimization in relational databases: A survey. *Statistical Analysis and Data Mining*, 1(4):223–250, 2009. 25

[46] D. Harish, Pooja N. Darera, and Jayant R. Haritsa. On the production of anorexic plan diagrams. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, VLDB '07, pages 1081–1092, 2007. 13, 32, 52, 62, 77, 80, 92

[47] D. Harish, Pooja N. Darera, and Jayant R. Haritsa. Identifying robust plans through plan diagram reduction. *Proc. of VLDB Endow.*, 1(1):1124–1140, 2008. 7, 21, 22, 26, 62, 69, 86

[48] Max Heimel, Martin Kiefer, and Volker Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proc. of the 2015 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '15, pages 1477–1492, 2015. 20

[49] Herodotos Herodotou and Shivnath Babu. Xplus: A sql-tuning-aware query optimizer. *Proc. of VLDB Endow.*, 3(1-2):1149–1160, September 2010. 21

[50] Yannis Ioannidis. The history of histograms (abridged). In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, VLDB '03, pages 19–30, 2003. 19

[51] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '91, pages 268–277, 1991. 5, 19

[52] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '98, pages 106–117, 1998. 7, 21, 22, 61

[53] Carl-Christian Kanne and Guido Moerkotte. Histograms reloaded: The merits of bucket diversity. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '10, pages 663–674, 2010. 19

[54] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997. 49

[55] Robert Philip Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, Cleveland, OH, USA, 1980. AAI8109596. 19

[56] Curtis Kroetsch. Multi-dimensional data statistics for columnar in-memory databases. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '14, pages 1605–1606, 2014. 19

[57] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung. Multi-dimensional selectivity estimation using compressed histogram information. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '99, pages 205–214, 1999. 19

[58] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. of VLDB Endow.*, 9(3):204–215, November 2015. 5, 18

[59] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. Adaptively reordering joins during query execution. In *Proc. of the 23rd Intl. Conf. on Data Engg.*, ICDE '07, pages 26–35, 2007. 21

[60] Lipyeow Lim, Min Wang, and Jeffrey Scott Vitter. SASH: A self-adaptive histogram set for dynamically changing workloads. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, VLDB '03, pages 369–380, 2003. 19

[61] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '90, pages 1–11, 1990. 7, 20

[62] Guy Lohman. Is query optimization a solved problem? http://wp.sigmod.org/?author=20, April 2014. 3, 5, 18, 86, 115

[63] Zvi Lotker, Boaz Patt-Shamir, and Dror Rawitz. Rent, lease or buy: Randomized algorithms for multislope ski rental. In *Proc. of the 25th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '08, pages 503–514, 2008. 45

[64] Clifford A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *Proc. of the 14th Intl. Conf. on Very Large Data Bases*, VLDB '88, pages 240–251, 1988. 19

[65] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '04, pages 659–670, 2004. 5, 21, 22, 25, 86, 118

[66] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '98, pages 448–459, 1998. 19

[67] Chaitanya Mishra and Nick Koudas. Join reordering by join simulation. In *Proc. of the 25th Intl. Conf. on Data Engg.*, ICDE '09, pages 493–504, 2009. 21

[68] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Boehm. Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in sap hana. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '14, pages 361–372, 2014. 19

[69] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. of VLDB Endow.*, 2(1):982–993, August 2009. 7, 18, 19, 22, 26

[70] M. Muralikrishna and David J. DeWitt. Equi-depth multidimensional histograms. In *Proc. of the 1988 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '88, pages 28–36, 1988. 19, 20

[71] Sumit Neelam. Design and implementation techniques for plan bouquet. Master's thesis, Database Systems Lab, Indian Institute of Science, Bangalore, 2014. 78

[72] Thomas Neumann and César A. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *BTW 2013*, pages 73–92, 2013. 7, 21, 22, 26

[73] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proc. of the 31st ACM Symposium on Principles of Database Systems*, PODS '12, pages 37–48, 2012. 25

[74] Frank Olken and Doron Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 5(1):25–42. 7, 20

[75] Aditya Parameswaran. An interview with surajit chaudhuri. *XRDS*, 19(1):38–39, September 2012. 3

[76] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '84, pages 256–276, 1984. 19

[77] Neoklis Polyzotis. Selectivity-based partitioning: A divide-and-union paradigm for effective query optimization. In *Proc. of the 14th ACM Intl. Conf. on Information and Knowledge Management*, CIKM '05, pages 720–727, 2005. 21, 27

[78] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, VLDB '97, pages 486–495, 1997. 19

[79] C. Rajmohan. Turbocharging plan bouquet identification. Master's thesis, Database Systems Lab, Indian Institute of Science, Bangalore, 2015. 63, 69

[80] Christopher Ré and Dan Suciu. Understanding cardinality estimation using entropy maximization. In *Proc. of the 29th ACM Symposium on Principles of Database Systems*, PODS '10, pages 53–64, 2010. 24

[81] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *Proc. of the 31st Intl. Conf. Very Large Data Bases*, VLDB '05, pages 1228–1239, 2005. 18, 32

[82] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '79, pages 23–34, 1979. 1, 5, 19

[83] Joshua Spiegel and Neoklis Polyzotis. Graph-based synopses for relational selectivity estimation. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '06, pages 205–216, 2006. 19

[84] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, VLDB '01, pages 19–28, 2001. 5, 7, 20

[85] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal*, 22(1):3–27, February 2013. 5, 19, 24

[86] Marianne Winslett. David dewitt speaks out: On rethinking the cs curriculum, why the database community should be proud, why query optimization doesn't work, how supercomputing funding is sometimes very poorly spent, how he's not a good coder and isn't smart enough to do db theory, and more. *SIGMOD Rec.*, 31(2):50–62, June 2002. 3

[87] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigms, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Proc. of 29th Intl. Conf. on Data Engg.*, ICDE '13, pages 1081–1092, 2013. 5, 78, 115

[88] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. CS2: a new database synopsis for query estimation. In *Proc. of the 2013 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '13, pages 469–480, 2013. 20

12.A Appendix

12.A.1 Query Text (based on benchmark queries)

```
select
        n_name,
        l_extendedprice * (1 - l_discount) as revenue
from
        customer, orders, lineitem, supplier, nation, region
where
        c_custkey = o_custkey and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and s_nationkey = n_nationkey and n_regionkey = r_regionkey
        and o_orderdate >= 1994-01-01
        and o_orderdate < 1994-01-01 + interval '25' day
        and c_acctbal <= 9900 and s_acctbal <= 9900
```

Figure 12.1: 3D_H_Q5 (Based on TPC-H Query 5)

```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer, orders, lineitem, supplier, nation, region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'
        and o_totalprice ≤ $X1
        and c_acctbal ≤ $X2
        and l_extendedprice ≤ $X3
group by
        n_name
order by
        revenue desc
```

Figure 12.2: 3D_H_Q5^b (Based on TPC-H Query 5)

```
select
        supp_nation, cust_nation, l_year, volume
from (
        select
                n1.n_name as supp_nation, n2.n_name as cust_nation,
                extract(year from l_shipdate) as l_year, l_extendedprice * (1- l_discount)
                as volume
        from
                supplier,lineitem, orders, customer, nation n1, nation n2
        where
                s_suppkey = l_suppkey and o_orderkey = l_orderkey
                and c_custkey = o_custkey and s_nationkey = n1.n_nationkey
                and c_nationkey = n2.n_nationkey
                and l_shipdate between date '1995-01-01' and date '1996-12-31'
                and c_acctbal <= 9900 and s_acctbal <= 9900 )
```

Figure 12.3: 3D_H_Q7 (Based on TPC-H Query 7)

```
select
        supp_nation, cust_nation, l_year, volume
from (
        select
                n1.n_name as supp_nation, n2.n_name as cust_nation,
                extract(year from l_shipdate) as l_year, l_extendedprice * (1- l_discount)
                as volume
        from
                supplier,lineitem, orders, customer, nation n1, nation n2
        where
                s_suppkey = l_suppkey and o_orderkey = l_orderkey
                and c_custkey = o_custkey and s_nationkey = n1.n_nationkey
                and c_nationkey = n2.n_nationkey
                and ( (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
                or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE') )
                and l_shipdate between date '1995-01-01' and date '1996-12-31'
                and c_acctbal <= 9900 and s_acctbal <= 9900 )
```

Figure 12.4: 5D_H_Q7 (Based on TPC-H Query 7)

```
select
        o_year, volume
from (
        select
                extract(year from o_orderdate) as o_year, l_extendedprice *
                (1-l_discount) as volume, n2.n_name as nation
        from
                part, supplier, lineitem, orders, customer, nation n1, nation n2, region
        where
                p_partkey = l_partkey and s_suppkey = l_suppkey
                and l_orderkey = o_orderkey and o_custkey = c_custkey
                and c_nationkey = n1.n_nationkey and n1.n_regionkey = r_regionkey
                and s_nationkey = n2.n_nationkey
                and o_orderdate between date '1995-01-01' and date '1995-09-01'
                and p_type = 'ECONOMY ANODIZED STEEL'
                and c_acctbal <= 9900 and s_acctbal <= 9900 )
```

Figure 12.5: 4D_H_Q8 (Based on TPC-H Query 8)

```
select
    o_year,
    sum(case when nation = 'BRAZIL' then volume
    else 0 end) / sum(volume) as mkt_share
from
    (
    select
        DATE_PART('YEAR',o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) as volume,
        n2.n_name as nation
    from
        part, supplier, lineitem, orders,
        customer, nation n1, nation n2, region
    where
        p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = 'AMERICA'
        and s_nationkey = n2.n_nationkey
        and p_retailprice ≤ $X1
        and s_acctbal ≤ $X2
        and l_extendedprice ≤ $X3
        and o_totalprice ≤ $X4
    ) as all_nations
group by
    o_year
order by
    o_year
```

Figure 12.6: 4D_H_Q8^b (Based on TPC-H Query 8)

```
select
        i_item_id, ss_quantity, ss_list_price,
        ss_coupon_amt, ss_sales_price
from
        store_sales, customer_demographics, date_dim, item, promotion
where
        ss_sold_date_sk = d_date_sk and ss_item_sk = i_item_sk and
        ss_cdemo_sk = cd_demo_sk and ss_promo_sk = p_promo_sk and
        cd_gender = 'F' and cd_marital_status = 'M' and cd_education_status = 'College'
        and (p_channel_email = 'N' or p_channel_event = 'N') and d_year = 2001
        and i_current_price < 99 and p_cost <= 1000
```

Figure 12.7: 4D_DS_Q7 (Based on TPC-DS Query 7)

```
select
        ca_zip, cs_sales_price
from
        catalog_sales, customer, customer_address, date_dim
where
        cs_bill_customer_sk = c_customer_sk and c_current_addr_sk = ca_address_sk
        and ( substr(ca_zip,1,5) in ('85669', '86197','88274', '83405',
        '86475', '85392', '85460', '80348', '81792')
        or ca_state in ('CA','WA','GA'))
        and cs_sold_date_sk = d_date_sk and d_moy = 2 and d_year = 1999
```

Figure 12.8: 3D_DS_Q15 (Based on TPC-DS Query 15)

```
select
        i_brand_id brand_id, i_brand brand, i_manufact_id,
        i_manufact, ss_ext_sales_price
from
        date_dim, store_sales, item, customer, customer_address, store
where
        d_date_sk = ss_sold_date_sk and ss_item_sk = i_item_sk
        and i_manager_id=97 and d_moy=12 and d_year=2002
        and ss_customer_sk = c_customer_sk and c_current_addr_sk
        =ca_address_sk and substr(ca_zip,1,5) <> substr(s_zip,1,5)
        and ss_store_sk = s_store_sk
        and s_tax_percentage <= 0.1
```

Figure 12.9: 5D_DS_Q19 (Based on TPC-DS Query 19)

```
select
        i_item_id, avg(cs_quantity) , avg(cs_list_price) ,
        avg(cs_coupon_amt) , avg(cs_sales_price)
from
        catalog_sales, customer_demographics, date_dim, item, promotion
where
        cs_sold_date_sk = d_date_sk and cs_item_sk = i_item_sk and
        cs_bill_cdemo_sk = cd_demo_sk and cs_promo_sk = p_promo_sk
        and cd_gender = 'F' and cd_marital_status = 'U' and cd_education_status
        = Unknown' and (p_channel_email = 'N' or p_channel_event = 'N') and
        d_year = 2002 and i_current_price <= 99
group by
        i_item_id
order by
        i_item_id
```

Figure 12.10: 4D_DS_Q26 (Based on TPC-DS Query 26)

```
select
        cc_call_center_id , cc_name , cc_manager , sum(cr_net_loss)
from
        call_center,catalog_returns, date_dim, customer, customer_address,
        customer_demographics, household_demographics
where
        cr_call_center_sk = cc_call_center_sk and cr_returned_date_sk =
        d_date_sk and cr_returning_customer_sk= c_customer_sk and cd_demo_sk
        =c_current_cdemo_sk and hd_demo_sk = c_current_hdemo_sk and
        ca_address_sk = c_current_addr_sk and d_year = 2000 and d_moy = 12
        and ( (cd_marital_status = 'M' and cd_education_status = 'Unknown')
        or(cd_marital_status = 'W' and cd_education_status = 'Advanced Degree'))
        and hd_buy_potential like '5001-10000%' and ca_gmt_offset = -7

group by
        cc_call_center_id,cc_name,cc_manager,cd_marital_status, cd_education_status

order by
        sum(cr_net_loss) desc
```

Figure 12.11: 4D_DS_Q91 (Based on TPC-DS Query 91)

```
select   s_store_name, hd_dep_count, ss_list_price, s_company_name
from
        store_sales, household_demographics, time_dim, store
where
        ss_sold_time_sk = time_dim.t_time_sk and
        ss_hdemo_sk = household_demographics.hd_demo_sk and
        ss_store_sk = s_store_sk and time_dim.t_hour = 8
        and time_dim.t_minute >= 30 and
        household_demographics.hd_dep_count = 2
        and store.s_store_name = 'ese'
```

Figure 12.12: 3D_DS_Q96 (Based on TPC-DS Query 96)