Geometric Search Techniques
for Provably Robust Query Processing

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Srinivas Karthik V.

Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

December, 2019
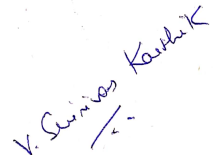
Declaration of Originality

I, **Srinivas Karthik V.**, with SR No. **04-04-00-10-12-13-10465** hereby declare that the material presented in the thesis titled

Geometric Search Techniques for Provably Robust Query Processing

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

- I have not manipulated any of the data or results.

- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.

- I have explicitly acknowledged all collaborative research and discussions.

- I have understood that any false claim will result in severe disciplinary action.

- I have understood that the work may be screened for any form of academic misconduct.

Date: 16-Dec-2019   Student Signature

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

Advisor Name: Prof. Jayant R. Haritsa   Advisor Signature

DEDICATED TO

The Supreme Power

- The God himself

Acknowledgements

I would like to be extremely thankful to the God, my Guru - Jagadguru Sri Sri Bharathi Teertha Mahaswami, and my family for everything in principle. My parents' and brother's (Dr. Srinivas Vivek) academic inclination and constant encouragement propelled me to take up the PhD programme and reach this stage. I am in-debt to their sacrifices, love they have showered on me and invaluable support throughout my academic journey. My brother has been a huge source of motivation to me, in short, a great friend-philosopher-guide. I always enjoy the endless discussions with him on philosophy of life and academics.

I am immensely grateful to my advisor Prof. Jayant R. Haritsa for his guidance. I am sure this thesis would have not been possible without his support. Further, no words can express my gratitude towards him, and will remain specially thankful for his help during important phases. It has been a great experience working with him and feel privileged for having got a glimpse of his vast knowledge and experience. Meetings with him has always been a pill of inspiration. His work ethics, ability to raise the standards of students, humility and sense of humor is something that I am fond of, and would like to imbibe them to a great extent possible.

It also has been a fantastic experience working with Dr. Vinayaka Pandit even from my IBM days. He has been a great mentor for me both personally and professionally. Needlessly to say that his support has shaped me as a much better researcher, and immensely thankful for introducing me to Prof. Haritsa that eventually led me to pursue this thesis under his guidance. Dr. Sreyash Kenkre, my another IBM colleague, has been very supportive throughout my thesis. Technical interactions with him is something I have always cherished.

During my PhD, one of the wonderful moments personally was my marriage with Dr. Shamaa Kamalanathan, and birth of our son Vivaswath. I am grateful to my wife for playing a very important role during the second half of my PhD. The arrival of my son has bought a positive change in our life, and special thanks to him for making my PhD as joyful as possible. Special thanks to my aunt Shobhamani, in-laws, Sumana, Seema, Samanvitha and my cousins.

Needless to say that all my lab mates has immensely helped me in several aspects. Primarily, technical discussions with Dr. Anshuman Dutt has played crucial role in this thesis. It was really a

i

Abstract

Relational Database Management Systems (RDBMS) constitute the backbone of today's information-rich society, providing a congenial environment for handling enterprise data during its entire life cycle of generation, storage, maintenance and processing. The Structured Query Language (SQL) is the de facto standard interface to query the information present in RDBMS based repositories. An extremely attractive feature of SQL is that it is "declarative" in nature, meaning that the user specifies only the end objectives, leaving to the system the task of identifying the optimal execution strategy to achieve these objectives.

A crucial input to generating efficient query execution strategies, called "plans", are the statistical estimates of the output data volumes for the algebraic predicates present in the query. However, in practice, these estimates, called "selectivities", are often significantly in error with respect to the actual values subsequently encountered during query execution. These inaccuracies arise due to a variety of reasons including intrinsic database complexities such as data skew and correlations. The unfortunate outcome is a poor choice of execution plan, leading to highly inflated query response times. Therefore, achieving robustness in query execution performance has been a long-standing open problem in the database research community.

The first success in the above quest was achieved five years ago by the `PlanBouquet` algorithm. The algorithm provide guarantees on *Maximum Sub-optimality (MSO)*, a metric capturing the worst-case execution performance relative to an oracular system that magically knows the correct selectivities. The guarantees are achieved by constructing a carefully calibrated "trial-and-error" sequence of time-budgeted plan executions that lead to run-time selectivity discovery in a space constructed from the error-prone predicates. However, in spite of this breakthrough, `PlanBouquet` suffers from critical limitations, including: (i) huge compile-time efforts to be amenable for run-time MSO guarantees, (ii) inability to handle queries with a large number of predicates prone to estimation errors, and (iii) performance variability across database platforms.

In this thesis, we address all the above-mentioned issues and take a substantive step forward in delivering practical robust query processing. Specifically, we design a new suite of robust query processing algorithms based on potent geometrical search techniques. We begin with `SpillBound`,

which provides an MSO guarantee of $D^2 + 3D$, where D is the number of predicates prone to estimation errors. This is achieved by incorporating focused allocation of execution time budgets through "spilling", whereby operator pipelines are prematurely terminated at chosen locations in the plan tree. The spilling feature extends `PlanBouquet`'s hypograph pruning of the selectivity space to a much stronger half-space pruning. A collateral benefit is that the guarantee is platform-independent and can be issued simply by query inspection. Further, we also prove a lower bound of D on the guarantee, which shows that `SpillBound` is within a factor of $\mathcal{O}(D)$ of the best deterministic algorithm in its class. Through an optimized variant of `SpillBound`, called `AlignedBound`, we achieve the linear lower bound for a restricted set of environments. Finally, when empirically evaluated on contemporary database engines over both synthetic and real-world benchmarks, `SpillBound` and `AlignedBound` provide markedly superior robustness as compared to `PlanBouquet`, with `AlignedBound` typically achieving single-digit MSO guarantees.

Although providing strong and portable performance guarantees, `SpillBound` falls prey to the "curse of dimensionality" since the query compilation overheads are exponential in D, and contemporary decision-support queries often possess a high ab initio value for this parameter. We tackle this issue in the second segment of the thesis by proposing a principled and efficient three-stage pipeline, called Dimensionality Reduction, that reduces the effective D to "anorexic" (small absolute number) levels even for highly complex queries. This drastic reduction results in `SpillBound` becoming practical for canned queries that are repeatedly invoked by the parent application. However, for ad-hoc queries which are issued on the fly, the overheads prove to be still too high. Therefore, in the third segment of the thesis, we investigate the trade-off between compilation overheads and the MSO guarantees. This leads us to a modified version of `SpillBound`, called `FrugalSpillBound`, that explicitly leverages the concave-down trajectory typically exhibited by plan cost functions over the selectivity space. From a theoretical perspective, `FrugalSpillBound` exponentially reduces the compilation overheads for a linear increase in the MSO guarantee. Empirically, we obtain more than three orders of magnitude reduction in exchange for a doubling in MSO.

The above robustness guarantees are welcome for many real-world query workloads that exhibit error-prone selectivity estimation. However, it is possible that even these workloads may include specific database queries for which the native query optimizer itself is capable of accurate selectivity estimations, and hence efficient plan choices with an MSO close to 1. For such queries, our query processing algorithms would be an overkill, incurring unnecessary performance penalties. Therefore, in the last segment of the thesis, we construct a software assist, called `OptAssist`, that aids the user in making the choice of whether to use the native optimizer or our robust alternatives.

Abstract

Overall, in this thesis, we achieve theoretical and practical performance guarantees for SQL query processing by leveraging a potent set of geometrical search techniques, thereby taking a major step towards making robust query processing a contemporary reality.

Publications based on this Thesis

1. Srinivas Karthik, Jayant R. Haritsa, Sreyash Kenkre and Vinayaka D. Pandit
   Platform-independent Robust Query Processing,
   In *Proc. of the 32nd Intl. Conf. on Data Engg., ICDE '16*, pages 325-336, 2016.
   (Best Student Paper Award).

2. Srinivas Karthik
   Robust Query Processing,
   In *PhD Workshop, Proc. of the 32nd Intl. Conf. on Data Engg., ICDE '16*, pages 226-230, 2016.

3. Srinivas Karthik, Jayant R. Haritsa, Sreyash Kenkre and Vinayaka D. Pandit
   A Concave Path to Low-overhead Robust Query Processing,
   In *Proc. of the VLDB Endow., 11(13)*, pages 2183-2195, 2018.

4. Sanket Purandare, Srinivas Karthik and Jayant R. Haritsa
   Dimensionality Reduction Techniques for Robust Query Processing,
   Technical Report TR-2018-02, DSL CDS/CSA, IISc, 2018.
   dsl.cds.iisc.ac.in/publications/report/TR/TR-2018-02.pdf.

5. Srinivas Karthik, Jayant R. Haritsa, Sreyash Kenkre, Vinayaka D. Pandit and Lohit Krishnan
   Platform-independent Robust Query Processing,
   In *IEEE Trans. on Knowledge and Data Engg. (TKDE), 31(1)*, pages 17-31, 2019.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database Management Systems (DBMS) typically model data in the form of tables, called relations, and query the information using the Structured Query Language (SQL). SQL is *declarative* in nature and hence specifies *what* has to be done, and not *how* to do it. Let us consider a simple example SQL query on the TPC-DS benchmark database as shown in Figure 1.1 (a). The query lists details of order sales from stores that are sold in a particular year.

(a) Example Query (EQ)   (b) Optimizer Chosen Plan for EQ
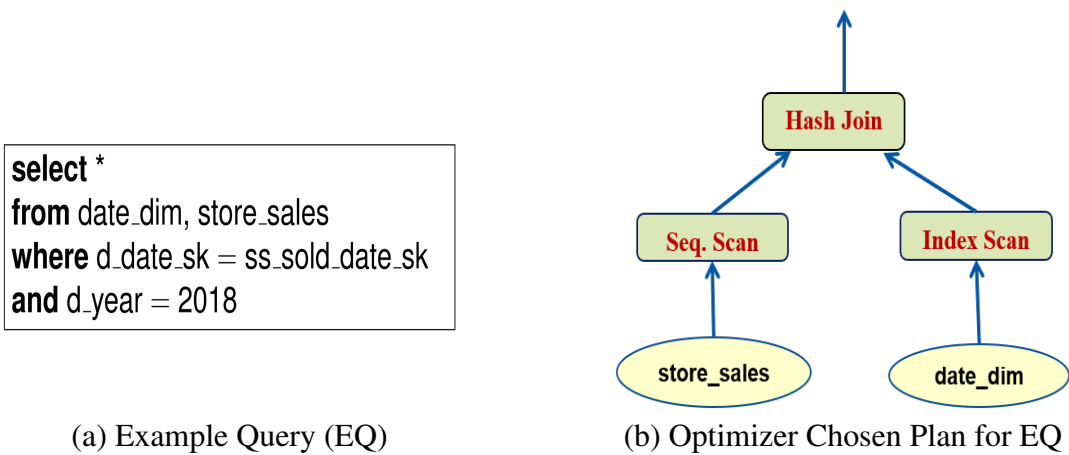
Figure 1.1: Example Query and an Optimizer Chosen Plan

Given a query, there exist a large number of semantically equivalent *plans* that can be used to process the query, which are exponential in the number of *relations*. Associated with each plan is its *cost*, which is typically a measure of its expected query response time. Since, the cost difference between the best execution plan and a randomly chosen one can be very high, database systems incorporate a *query optimizer* module to automatically find the best query execution strategy. Figure 1.1 (b) captures the best plan for the SQL query listed in Figure 1.1 (a), as chosen by an optimizer. The plan uses

1

Sequential scan access path for store_sales, and Index scan for date_dim. Finally, they are joined using the Hash join algorithm.

1.1 Query Optimizer Framework

Let us now see the framework used by modern query optimizers[1]. These query optimizers each have their own "secret sauce" to identify the best (i.e. cheapest/fastest) plan for answering declarative SQL queries. However, the de-facto standard underlying strategy present in all these systems, which was pioneered by the System R project at IBM Research [SAC⁺79], is the following:

1. First, apply a variety of heuristics to restrict the exponential plan search space to a manageable size. For instance, the early database systems only considered left-deep plan trees.

2. Next, estimate with a cost model and a dynamic-programming-based processing algorithm, the efficiency of each of these candidate plans.

3. Finally, choose the plan with the lowest estimated cost.
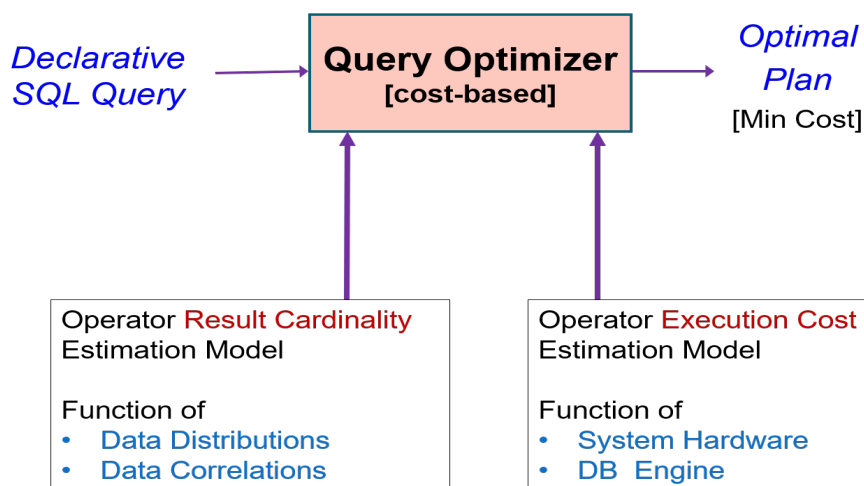
Figure 1.2: Query Optimizer Framework

A pictorial representation of the canonical query optimization framework is shown in Figure 1.2. The input is the declarative SQL query and the output is the *optimal* (i.e. cheapest/fastest) execution plan. Its core processing is mostly implemented by the dynamic programming-based search algorithm, which leverages the fact that the globally optimal plan can be incrementally built up using

---

[1]Some of the contents in the rest of the chapter, corresponding to standard material in query optimization, are borrowed from [Har18] with the author's consent.

the locally optimal solution for each operator. There are two essential models that serve as inputs to this process, namely, the *cardinality estimation* and the *cost estimation* models, which are described below.

1. *Cardinality Estimation Model:* This model estimates the volume of data, measured in number of database rows, that flows from one operator to the next in the plan tree. For the example query, the cardinalities are estimated for the filter predicate (d_year = 2018) and the join predicate (date_dim ⋈ store_sales), as captured in red coloured text in Figure 1.3 (a). Further, 365 out of 7.3×10^5 tuples are estimated to pass through the (d_year = 2018) predicate. These estimates are a function of the data distributions within the relational columns, and the data correlations across the columns. The individual column distributions are usually approximated, in a piece-wise manner, using histogram-based summaries.

2. *Cost Estimation Model:* This model is responsible for estimating the time taken for processing the data at each operator in the plan. As shown in Figure 1.2, its estimates, which are usually computed on a normalized per-row basis, are dependent on the underlying computing platform and the software implementation of the database engine. The overall cost of each operator is the product of the estimated row cardinalities (as obtained from the cardinality model) and the per-row cost estimates.

1.2 Optimizer Challenges

In practice, both the cardinality and cost estimation models are erroneous, often leading to poor choices of query execution plans [Loh14]. In particular, the estimates in cardinality model are often significantly in error with respect to the actual values subsequently encountered during query execution. Such errors, which can even be in orders of magnitude in real database environments [MRS+04, Loh14, LGM+15], arise due to a variety of well-documented reasons [SLMK01, Loh14], including outdated statistics, coarse summaries, attribute value independence assumptions, complex user-defined predicates, and error propagation in the query plan tree [IC91]. Moreover, in industrial environments such as ETL (Extract, Transform, Load) workflows, the statistics may actually be unavailable due to data source constraints, forcing the optimizer to resort to "magic numbers" for the cardinality values (e.g. 0.1R for equality selections on columns of a relation with R rows [SAC+79]).

Example cardinality estimation errors for our example query, corresponding to huge overestimates, as can be seen in Figure 1.3 (a). The query was run over a slightly modified version of 100 GB TPC-DS database, wherein less than 10 tuples were inserted to the original data. In the figure, the cardinalities shown in red colour map to the estimated cardinalities whereas the one shown in green

colour correspond to the actual cardinalities. For instance, the estimated cardinality of the query output is 13 million, while the actual is just 1! However, if we magically had got the estimates correct, then the resulting optimal plan is shown in Figure 1.3 (b). The estimation errors originated from the filter equality predicate (d_year = 2018) wherein the optimizer estimated cardinality for the predicate (i.e., 365) was based on the invalid uniform data distribution assumption. Then the error propagated to the upper join operator, with the error getting magnified again due to the uniformity assumption wrt primary key and foreign key join getting violated.
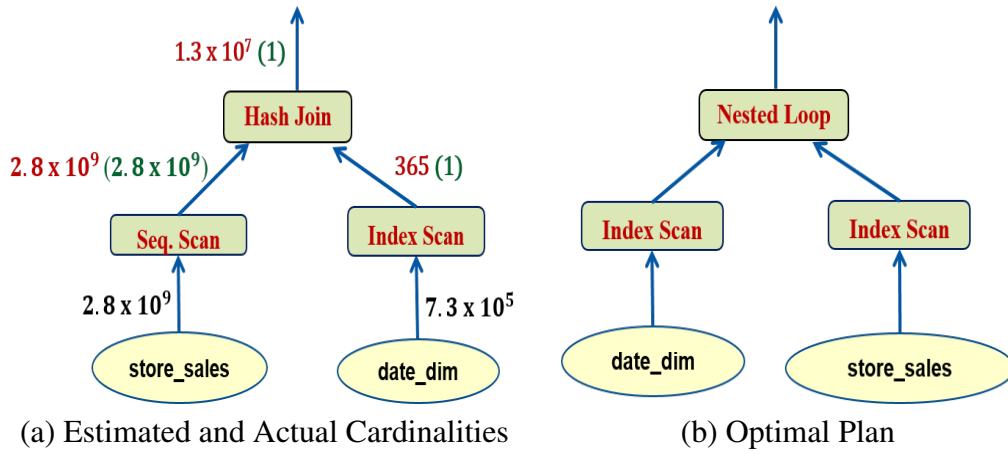
Figure 1.3: Example of Large Estimation Errors

The scale of performance degradation, in terms of query response time, faced by database queries arising out of the poor choices of execution plans, can be huge - often in *orders of magnitude* - as compared to an oracle that magically knows the optimal plan for processing the query. As a case in point, for our example query, the slowdown in query response time, under PostgreSQL, relative to the optimal plan's response time, exceeds *four orders of magnitude*! – the optimal plan took less than around 70 ms whereas the time taken by the optimizer chosen plan shot upto around 30 mins.

Finally, apart from the obvious negative impact on user productivity and satisfaction, there are also financial implications of this performance degradation – for instance, it is reported that the lack of robustness can contribute as much as a third to the total cost of ownership for a database system – primarily due to lost efficiency, over-provisioning of resources and increased human administrative costs [WKG09].

These challenges in query optimization are corroborated by the following statements from leading academic and industrial experts:

- *Dr. Surajit Chaudhuri, Microsoft Research (2009, 2012): 1) To be provocative, one can say that though the optimizers of today's relational databases are able to do surprisingly sophisticated*

optimization because of the power of transformation rules and their extensible framework; yet they have significant weaknesses that lead to unexpectedly poor selection of execution plans at times [Cha09]. 2) 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 [Par12].

- *Dr. Guy Lohman, IBM Research (2014): 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?"[Loh14] !*

- *Prof. Jeffrey Naughton, University of Wisconsin - Madison (2016): In database query evaluation, the difference between a good plan and a bad or even average plan can be multiple orders of magnitude - so successful query optimization makes the difference between a plan that runs quickly and one that never finishes at all. Accordingly, since the seminal papers in the 1970s, query optimization has received and continues to receive a great deal of attention from both the industrial and research database communities [Nau16].*

In a nutshell, in spite of the long-standing research in this area, the query processing domain has largely remained a *black art* and *fail to provide robust query performance*.

1.3 Robustness for Database Systems

The performance of database systems substantially depends on the accuracy of the underlying estimation models. Due to the inaccuracies in the model, heavy performance penalties are being paid. Therefore, it is highly desirable to design robust solutions that provide performance stability. The importance of robustness in query performance is also showcased by the fact that there have been no less than three Dagstuhl seminars on this topic [dag10, dag12, dag17] over the last decade.

Robust query processing for database systems can be seen as providing *good* and *predictable* performance, even in case of highly uncertain database environments. However, the definition of robustness itself has been debated for a long time, and no consensus has been achieved [dag10]. For instance, one or more of the following could be a notion of robustness [Har19]:

- Worst-case performance improvement possibility at the expense of average-case.

- Graceful performance degradation as opposed to "performance cliffs".

- Ability to seamlessly scale with work-load complexity, database size, distributional skew and correlations.

- Providing strong theoretical guarantees wrt an oracular ideal.

Perhaps, robustness encompasses all of these scenarios with the specific choices being application-dependent.

It is known that the cardinality model induces orders of magnitude errors, thus having huge impact on plan choice. However, the same is not true with the cost model wherein, in general, it induces relatively small errors thus having limited impact on plan choice [Loh14]. Hence, *this thesis focuses on mitigating the errors induced by the cardinality model with the aim of providing strong theoretical guarantees wrt an oracular ideal*.

1.4 Prior Work

A considerable body of literature exists on proposals to tackle this classical problem. The proposed techniques include: (a) Improving estimation accuracy through novel statistical models, sampling and execution-feedback mechanisms [AC99, MNS09, TDJ13]; (b) Identifying execution plans that are relatively less sensitive to estimation errors [CHG02, BC05, DDH08, WBM+18]; and (c) Dynamically changing plans at run-time if estimation errors are detected during the execution of the originally chosen plan [KD98, BBD05, NG13].

While all these prior techniques provide novel and innovative formulations, a common limitation is their inability to furnish performance guarantees. That is, we can only hope for good performance, but cannot provide *provable* bounds in this regard. A notable exception to the above mentioned prior literature is the `PlanBouquet` algorithm [DH16], proposed five years ago. The algorithm provide guarantees on **Maximum Sub-optimality (MSO)**, a metric capturing the worst-case execution performance relative to an oracular system that magically knows the correct selectivities.

The goal of this thesis is to design query processing algorithms that provide strong theoretical guarantees on the MSO metric. Given that the `PlanBouquet` technique provide bounded MSO guarantees, our work builds on this framework. Let us now see the precise definition of MSO followed by the brief description of the `PlanBouquet` algorithm.

1.4.1 Maximum Sub-Optimality – Robustness Metric

MSO of a query processing algorithm captures the worst-case ratio, over the entire modeled selectivity space, of its execution cost with respect to the optimal cost incurred by an oracular system that magically knows all the correct selectivity values. Here, selectivity refers to normalized cardinality which is a widely used terminology in database literature.

The precise mathematical definition of MSO is as follows: Assume that we have an multi-dimensional selectivity space, referred to as the *Predicate Selectivity Space* (PSS). Here, the dimensions in PSS correspond to predicates in the input query for which the optimizer estimation module is invoked. A sample PSS on two predicates X and Y, with an example estimation error, is shown in

Figure 1.4. The two axes in the figure correspond to the varying selectivities, respectively from $(0, 1]$, of the two predicates.
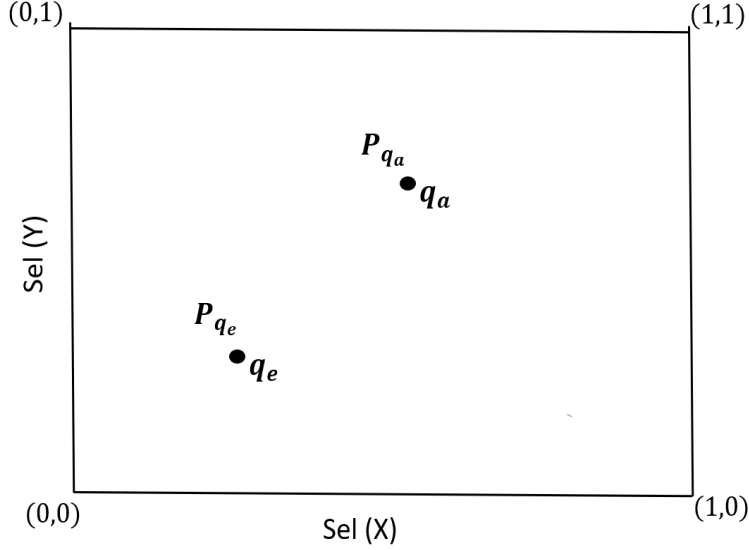
Figure 1.4: Sample 2D PSS

A traditional query optimizer will first estimate $q_e \in$ PSS, and then use P_{q_e} to execute the query. However, from execution we may infer that the actual query location to be at $q_a \in$ PSS. Let $Cost(P, q)$ represent the cost of executing a generic plan P at an arbitrary query location q. Then, the sub-optimality of using P_{q_e}, relative to an oracle that magically knows the actual location, and therefore uses the ideal plan P_{q_a}, is defined as:

$$SubOpt(q_e, q_a) = \frac{Cost(P_{q_e}, q_a)}{Cost(P_{q_a}, q_a)} \tag{1.1}$$

With this characterization of a specific (q_e, q_a) combination, the MSO that can potentially arise over the *entire* PSS is given by:

$$MSO = \max_{(q_e, q_a) \in \text{PSS}} (SubOpt(q_e, q_a)) \tag{1.2}$$

As per this formulation, MSO values range over the interval $[1, \infty)$. Then, *the objective is to design query processing algorithms which minimize MSO.*

1.4.2 Plan Bouquet

The PlanBouquet algorithm has two phases - namely, compilation and execution. In the compile-time phase, the algorithm constructs the D-dimensional PSS. The next step is to identify, through

repeated invocations of the query optimizer, the "parametric optimal set of plans" (POSP) that cover the entire selectivity space contained in the `PSS`. This overlay exercise is carried out at a discretized resolution r along each dimension of the `PSS`, incurring a total of $\theta(r^D)$ calls to the query optimizer.

At run-time, starting from the origin of the `PSS` and moving outwards in the space, a carefully chosen subset of POSP plans, called the "plan bouquet", is sequentially executed until one reaches completion, with each execution assigned a time limit equal to the plan's optimizer-assigned cost. By sequencing the plan executions and their time limits in a calibrated manner, the execution overheads entailed by this "trial-and-error" exercise can be bounded, irrespective of the query location in the space. In particular, it is shown that $MSO \leq 4 * |\text{plan bouquet}|$. A more precise bound is given later in Chapter 3.

1.5 PlanBouquet's Limitations

Notwithstanding `PlanBouquet`'s welcome robustness, the framework suffers from key limitations on both execution and compilation phases, as listed below:

- *Execution Phase:* The plan bouquet set, and hence the previously mentioned MSO guarantee, is a function of not only the query, but also the optimizer's behavioral profile over the underlying database platform (including data contents, physical schema, hardware configuration, etc.). As a result the guarantee value becomes highly variable, depending on the specifics of the current operating environment.

- *Compilation Phase:* In this phase, `PlanBouquet` essentially needs to construct the `PSS` in order to be amenable for MSO guarantees in the execution phase. These efforts are exponential in the `PSS` dimensionality. This leads us to the following issues:

  (a) *High Dimensional Queries:* Since the contemporary OLAP queries often have high ab initio dimensions, the time consumed in this phase becomes *impractical* after six dimensions, even on contemporary servers.

  (b) *Ad-hoc Queries:* For queries upto to six dimensions, `PlanBouquet`'s compilation efforts are manageable for *canned* queries, which are repeatedly invoked by the parent application. However, for *ad-hoc* queries that are issued on the fly, the overheads prove to be still too high.

  (c) *Computation of Guarantee Value:* As the MSO guarantee are a function of the cardinality of the plan bouquet set, even just computing the guarantee value, let alone the execution, requires substantial investments in preprocessing overheads.

1.6 Our Contributions

In this thesis, we address all the above-mentioned issues and take a substantive step forward in delivering practical robust query processing. Specifically, we design a new suite of robust query processing algorithms based on a potent set of geometrical search techniques. Our contributions in this thesis, in terms of the enhancements over `PlanBouquet` in execution and compilation phases, are described next.

1.6.1 Execution Phase Enhancements

Our objective here is to develop a robust query processing approach that offers a strong MSO bound which is *solely query-dependent*, irrespective of the underlying database platform. That is, we desire a "structural bound" instead of a "behavioral bound".

SpillBound

With the above goal, we present a new query processing algorithm, called `SpillBound`, that achieves this objective in the sense that it delivers an MSO bound that is only a function of D, the *number* of predicates in the query that are prone to selectivity estimation errors. Moreover, the dependency is in the form of a low-order polynomial, with MSO expressed as $\mathbf{D^2 + 3D}$. Consequently, the bound value becomes: (i) independent of the database platform under the assumption that D remains constant across the platforms, (ii) known upfront by merely inspecting the query, and not incurring any compilation overheads, and (iii) certifiably low in value. These benefits are attained through executing plans in *spill-mode* wherein the operator pipelines are deliberately terminated at specific nodes of a plan tree. The spilling feature extends `PlanBouquet`'s hypograph pruning of the selectivity space to a much stronger half-space pruning.

Our experiments, under PostgreSQL, indicate that for most part, `SpillBound` provide similar guarantees to `PlanBouquet`, and occasionally, tighter bounds. More pertinently, the *empirical* MSO of `SpillBound` is significantly better than that of `PlanBouquet` for *all* the queries. As a case in point, for TPC-DS Query 26 with 4 error-prone predicates, the MSO guarantee is close to 40 with `PlanBouquet`, but comes down to 28 with `SpillBound`. With regard to empirical MSO, the value decreases from `PlanBouquet`'s 30.6 to 13 for `SpillBound`. The `SpillBound` algorithm and its experimental results are presented in Chapter 4.

LowerBound

At this juncture, a natural question to ask is whether some alternative selectivity discovery algorithm can provide better MSO bounds than `SpillBound`. In this regard, we prove that *no* deterministic technique can provide an MSO bound less than D. Therefore, the `SpillBound` guarantee is no

worse than a factor $\mathcal{O}(D)$ as compared to the best possible algorithm in its class.

AlignedBound

Given this quadratic-to-linear gap on the MSO guarantee, we seek to characterize scenarios in which `SpillBound`'s MSO approaches the lower bound by exploiting the geometric locations of the plans along the boundary of the selectivity space. This leads us to the design of `AlignedBound` algorithm that delivers an MSO in the platform-independent range $[\mathbf{2D + 2, D^2 + 3D}]$. Further, its empirical performance is typically closer to the *lower end* of its range, i.e. $2D+2$, and often provides substantial benefits for query instances that are challenging for `SpillBound`. For instance, `AlignedBound` brings the MSO of the previously-mentioned Q26 test case down to 9.2. Moreover, in the absolute sense, `AlignedBound` consistently collapses the enormous MSOs which are incurred with contemporary industrial-strength query optimizers, down to *single digit MSO guarantees*. The lower bound and `AlignedBound` algorithm with its empirical performance, are covered in Chapter 5.

1.6.2 Compilation Phase Enhancements

Moving our attention to compilation phase enhancements, our objective is to reduce the compilation efforts of `SpillBound` class of techniques so as to be amenable for *canned* and *ad-hoc* queries of high-dimensional selectivity spaces. We achieve this in two stages: Firstly, identify *removable* dimensions with the objective of minimizing compilation efforts while not violating the original MSO guarantees. Secondly, given the reduced set of dimensions, we (further) reduce the compilation efforts with moderate increase in MSO values.

Dimensionality Reduction

Although `SpillBound` class of techniques provide strong and platform independent guarantees, they suffer from the curse of dimensionality on two fronts – firstly, the overheads of constructing the `PSS` which are *exponential* in the number of its dimensions; and secondly, the MSO guarantees are *quadratic* in this dimensionality. Since contemporary OLAP queries often have more than 15 ab initio `PSS` dimensions, a legitimate question that arises is whether `SpillBound` can be made practical for current database environments.

In Chapter 6, we tackle this problem by presenting a principled and efficient *dimensionality reduction* process, called `DimRed`. This procedure incorporates a pipeline of reduction strategies whose collective benefits ultimately result in `PSS` dimensionalities that can be efficiently handled by modern computing environments. Further, the output of this process is the identification of *impactful* error-prone predicates which collectively form the Error Selectivity Space (`ESS`).

Our empirical results indicate that `DimRed` is consistently able to bring down the `PSS` dimensionality of the workload queries, some of which are as high as 19, to 5 or less. Further, not only the

preprocessing time taken reduces significantly, even the resulting MSOs are significantly better than those on the original system. In general, reductions are substantial enough to be useful in practice. As a case in point, for a query based on Q27 from TPC-DS, the original dimensionality of 9 is brought down to as low as 2. Further, the preprocessing time for $r = 100$, is reduced from *impractically high value (more than a year) to few hours*, and the MSO also saw a huge improvement from 108 to 20.

Frugal SpillBound

Even though `DimRed` algorithm provide substantial reduction in compilation overheads, a major limitation of `SpillBound` is that the reduced overheads are manageable for canned queries but still too high for ad-hoc queries.

With an objective of extending `SpillBound` to handle ad-hoc queries, in Chapter 7, we propose `FrugalSpillBound` algorithm which provably achieve significant reductions in the compilation overheads at the cost of mild relaxation on the MSO guarantees. The algorithm is designed leveraging an important observation that plan cost functions typically exhibit a *concave down* behaviour wrt predicate selectivities. The good news is that the tradeoff between the relaxation factor on the MSO guarantees and the relative reduction in compilation overheads is extremely attractive. Using the contemporary multi-core architectures, efficiency in handling of ad-hoc queries is increased by leveraging the inherent parallelism available in the `ESS` construction.

Our performance results indicate that, for twice relaxation in MSO guarantees, two orders of magnitude theoretical reduction in overheads is *routine*, and empirically, the benefits are more than *three orders of magnitude*. For instance, `FrugalSpillBound` brings the compilation overheads of the previously-mentioned Q27 from *few hours* to *few seconds* on a well-provisioned multi-core machine!

1.6.3 Deployment Aspects

An important aspect during the deployment of our technique is that there exist real-workloads which may include specific database queries for which the native query optimizer itself is capable of accurate selectivity estimations, and hence efficient plan choices with an MSO close to 1. For such queries, our query processing algorithms would be an overkill, incurring unnecessary performance penalties. In order to address this issue, we have built a software assist, `OptAssist`, which aids the user to choose the *better* of our proposed robust algorithms or the native optimizer for a given query instance.

The complete architecture of our proposed robust database engine in terms of the flow diagram is captured in Figure 1.5. The architecture has a driver layer that implements our proposed techniques by interacting with the underlying database engine. An input query first goes through the `DimRed` component to decide the set of removable dimensions. The query's impactful error-prone dimensions

are then fed into `OptAssist`, which helps the user to decide whether to choose a robust algorithm (either, `SpillBound` or `FrugalSpillBound` based on the query being respectively canned or ad-hoc) or the native optimizer. Once this is decided, if the native optimizer is chosen then the query results are produced using its plan choice. If not, based on the chosen robust algorithm, it goes through the compilation phase, i.e., generating the `ESS`. After this phase, the driver interacts with the executor module to finally produce the query results. Specifically, the driver layer, for every execution, chooses the plan and its execution time limit which is a function of *monitored selectivity* from previous executions. The `OptAssist` along with essential features required for deployment of our proposed robust database engine are presented in Chapter 8.
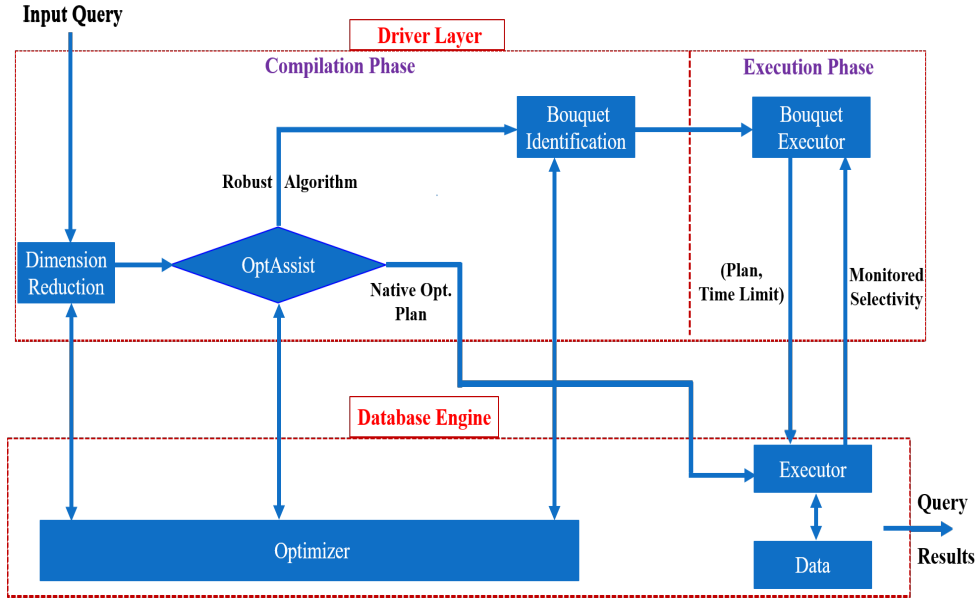
Figure 1.5: Architecture of Proposed Robust Database Engine

1.6.4 Summary

In this thesis, with the goal of achieving practical and provable SQL query processing, we build on the earlier proposed `PlanBouquet` framework. Specifically, we propose a suite of robust query processing algorithms that overcome the key limitations of the `PlanBouquet` framework by leveraging a potent set of geometrical search techniques. Overall, our proposed techniques corroborated with strong theoretical guarantees take a substantive step forward in practical robust query processing.

1.6.5 Thesis Organization

The remainder of the thesis is organized as follows: We start with reviewing the related literature in Chapter 2. This is followed by a formal description of the robust query processing problem, along

with the underlying assumptions and notations in Chapter 3. Platform-independent MSO guarantees are presented in Chapter 4. In the subsequent chapter, i.e. Chapter 5, lower bound results on MSO and an algorithm to achieve tighter MSO guarantees are described. Then, Chapter 6 discusses techniques to handle high dimension queries. Next, we move on to presenting algorithms to handle ad-hoc queries in Chapter 7. The `OptAssist` component for helping the user to choose between the better of our proposed robust query processing algorithm and the native optimizer, and the essential features required for the deployment of the proposed architecture are enumerated in Chapter 8. We conclude the thesis and discuss the future directions in Chapter 9.

Chapter 2

Related Work

In this chapter, we discuss the previous work related to robustness in database query processing. We start with presenting the prior work related to cardinality estimation and then describe remaining work in robust query optimization and processing.

Since the pioneering work of System R [SAC+79], there has been a plethora of work with respect to different aspects of query optimization. The approaches include improving the statistical quality of the meta-data through improved summary structures [MNS09, TDJ13], feedback-based adjustments [AC99], on-the-fly re-optimization of queries [KD98, BBD05, NG13] and identifying robust plans that are relatively less sensitive to estimation errors [CHG02, BC05, DDH08, WBM+18]. Although these approaches had novel formulations, they were unable to provide performance guarantees. As mentioned in the Introduction, the `PlanBouquet` framework [Dut] was the first work to provide bounded guarantees on worst-case performance wrt cardinality estimation errors – the current thesis builds on this work.

The related literature based on approaches to attack this chronic problem, as captured in Figure 2.1, can be broadly classified into: fully relying on selectivity estimation process to its partial dependence, and finally to no reliance on it. Our work and `PlanBouquet` share a significant fraction of our prior literature. Thus, here, we summarize the salient features of the common portion and request the reader to refer to [Dut, YHM15] for its details.

2.1 Full Dependence on Estimation Module

We now describe techniques that primarily try to improve the selectivity estimates, and then choose the best plan based on these estimates. In other words, the chosen plan's quality is primarily a function of the accuracy of the estimates.

Histograms and sampling are widely used approaches for selectivity estimation. Typically, the
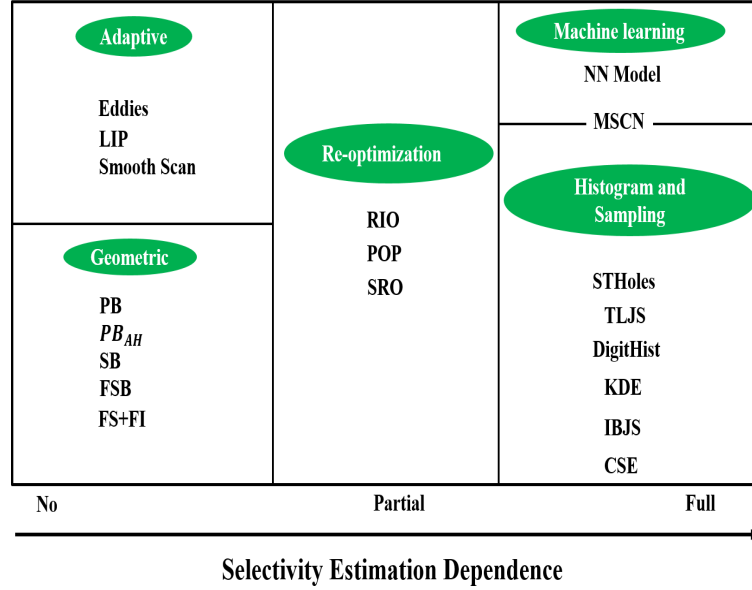
Figure 2.1: Approaches to tackle errors in Cardinality Model

eventual estimates are based on assumptions such as Attribute Value Independence (AVI) and certain uniformity of data distributions. Unfortunately, in practice none of these approaches can guarantee accurate estimates. However, one advantage of histogram based approaches is that, in certain scenarios, one can deduce deterministic upper and lower bounds on selectivities, while such bounds may not be feasible with non-histogram approaches.

A comprehensive survey on the standard estimation techniques is delineated in [Ioa03]. More recently proposed along this thread is the *DigitHist* technique [SDG17], which is a histogram summary for selectivity estimation. The key idea is to use multi-dimensional and its one-dimensional histograms along regular grids of different resolutions. Although their approach, while being restricted to filter predicates, improves the state-of-the-art accuracy. However, the experimental results still report nearly four orders of magnitude multiplicative (or relative) error in selectivity estimation quality, on an average, on certain datasets.

On the sampling front, recently Index-based join sampling (IBJS) for join estimation is proposed in [LRG$^+$17]. In this work, for a given sampling budget, the budget is spent in bottom-up manner to get the estimates of the intermediate results, and falls back to underlying engines estimates as the budget gets exhausted. Though the approach improves on the native optimizer's estimates on average, the paper still reports more than 10^6 multiplicative error in cardinality estimates in worst-case. Further, IBJS is heavily dependent on having suitable indexes. Combined Selectivity Estimation (CSE), a novel approach combines sampling with synopses for the purpose of estimating the selectivity of *conjunctive* predicates in [MMK18]. They only handle estimation of conjunctive filters of single

15

relation queries, and not that of standard benchmark queries involving multiple joins.

A two-level join sampling (TLJS) for join size estimation is proposed in [CY17], but their scheme works for at most two joins. Furthermore, their techniques require a sample size of around 1% for good accuracy. Similar is the case in [KHBM17] wherein Kernel Density Estimation (KDE) for estimating join selectivities. The evaluation of the proposed technique is performed for only upto three joins and, for 1% sample size, reports more than three orders of magnitude multiplicative errors on the estimation quality. The above mentioned techniques use a sample of around 1% of the underlying data size. Thus, running into *scalability* issues as the data grows, especially, in Big Data scenarios.

Machine Learning Techniques

After having seen some of the recent works on improving estimation using histograms and sampling, let us now look into the novel application of *Machine Learning Techniques* for the same. Here is an interesting quote made in [KYG$^+$18]: *Applications of machine learning to databases internals is still the subject of significant debate this year and will probably continue to be a contentious question for years to come*. One of the recent and initial works which try to explore the possible application of deep learning techniques to query optimization is [OBGK18]. They use neural networks (NN Model), and their work is in its stages showing results for estimation of only single join. Along similar lines, a multi-set convolutional network (MSCN) deep learning technique is applied for join cardinality estimation in [KKR$^+$19]. They also improve the state-of-the-art estimation accuracy, as well as addressing 0-tuple situations in case of base table samples. This scenario can happen in sampling based techniques when no tuples gets selected in base table samples due to very selective predicates. While their empirical evaluation is limited to four joins, they report around three orders of magnitude relative estimation errors in worst-case. Similar attempt is made in [DWN$^+$19], which uses neural networks to improve the estimation quality but the scope of their work is limited to multi-dimensional range predicates.

Remark 1: It has to be noted that there have been similar efforts in the past which try to improve the quality of histograms using feedback information such as *self-tuning histograms* (STHoles) [AC99, BCG01, KMSB15]. But inspite of these efforts we see large errors in the cardinality model. In general, the eventual model built using machine learning techniques inherently run into the issue of being predominantly dependent on historical training data. It is known that there exist scenarios where query performance is sensitive to minor changes in data. It is unlikely that current machine learning techniques capture these changes until they become part of training. Finally, currently none of these approaches are amenable to strong query performance guarantees. Hence, it would be an important future work for machine learning techniques to address the above issues.

Remark 2: The training phase of the machine learning techniques could be seen analogous to the initial compilation phase of our proposed robust algorithms. However, the outcome of our algorithms is stronger since we can provide guarantees on the worst-case query performance. Moreover the training requires a lot of thought about the choice of queries, etc., whereas ours is straightforward.

2.2 Partial Dependence on Estimation Module

In this category of techniques, the plan(s) used during execution are a function of both the optimizer compile-time estimates and the run-time monitoring of selectivities, thus *partially* depending on the estimation module.

Well known reoptimization techniques such as POP [MRS$^+$04] and Rio [BBD05] fall into this category of partial dependence on estimation module. They initially choose and execute a plan based on the estimates and then make an re-optimization if they find the estimates to be significantly different from the actual ones found during execution. Based on this information, a re-optimization may be triggered followed by execution of the new plan from scratch or using intermediate results. This whole process is continued until end of query execution. Both POP and Rio are based on heuristics and do not provide any performance bounds. Further, POP may get stuck with a poor plan since its selectivity 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 based solely on the performance at the corners of the PSS has not been validated.

One of the recent approaches along this line is the sampling-based query re-optimization (SRO) [WNS16]. Here, first the optimizer chosen plan (based on its estimates) is obtained, and then the plan is executed on sample data (instead of the original data) drawn independently from each table. Since the samples are independent, the sample size must be quite large to reduce the risk of empty join results (the paper uses 5% of the relation size). In every execution, the selectivity estimates are inferred which are then injected back into the query optimizer to compute a new query plan. The process is repeated until it converges to a plan, which is subsequently used during execution. As mentioned in [LRG$^+$17] sampling-based query re-optimization sometimes avoid bad query plans, but suffers from high sampling overheads, i.e., space and execution overheads.

2.3 No Dependence on Estimation Module

In this umbrella of techniques, the idea is to not depend on the selectivity estimates by understanding its limitations. In essence, the goal is to *aim for resistance rather than cure*. In the view point of resistance, the approaches include choosing plans which are *robust* to estimation errors. These robust approaches (as well as other approaches) that use a single plan during the entire query execution run into the basic infeasibility of a single plan being near-optimal across the entire selectivity space. On

the other end of the spectrum, there have been attempts to use different operators to execute different portions of the data [AH00]. While these are radically new attempts, they are limited in their ability to change join-order due to plan state management requirements.

Recently, there have been quite some works wrt no reliance on the estimation module. We present this literature by further dividing in terms of the techniques used to address them, which are primarily *geometric* or *adaptive*.

Geometry

Robustness metrics for plan selection wrt errors in cardinality estimation are proposed in [WBM$^+$18]. They essentially propose two themes of robustness metrics for any plan. The first one uses the slope of a plan using its plan cost function (PCF) and assigns a risk score, referred to by FS. The intuition behind this is that more the slope of a plan, more the risk, or equivalently, lesser is the plan robustness. In this work, a carefully chosen set of plans are given a FS score based on the corresponding PCF's slope at the optimizer estimated selectivities. Then the least FS score plan is chosen as the most robust plan and executed. Further in the second metric, integral value (i.e. area under the curve) of a plan using its PCF is taken as a robustness score of the plan, denoted by FI. Again, lesser the integral value more is the plan robustness. Here, the most robust plan chosen for execution is the least FI score plan from a carefully chosen set of plans. The evaluation of the algorithm show that these plan choices improve the execution performance from the native optimizer's plan choice by around a factor of two, on an average. However, they do not provide guarantees of their robust plan choice wrt *optimal* plan (i.e. error-free estimation scenario).

We have also empirically evaluated their technique and observed the following: Consider a simple query with just a filter predicate. Assuming the query is very selective to have index scan as the optimal plan. Since sequential has zero slope, the FS metric would suggest sequential scan as the most robust plan for the query. By this, we have a scenario wherein the sub-optimality of the plan chosen by FS metric, wrt optimal plan, is more than three orders of magnitude. Similarly, for the FI integral metric, a query with just one join and one filter predicate produced more than four orders of magnitude sub-optimal performance. These high sub-optimality examples can be extended for larger queries.

As mentioned before, `PlanBouquet` provide guarantees on worst-case execution performance and that our work builds on it. This leads us to the techniques presented in this thesis starting from `SpillBound` which achieves a platform-independent MSO guarantee of D^2+3D. Then to `DimRed` which handle high dimensional queries tackling the curse of dimensionality on both MSO guarantees and compilation overheads. Next, we present `FrugalSpillBound` which further reduce the compilation overheads to support ad-hoc. The limitation of huge compilation phase of `PlanBouquet`

is tried to mitigate in PB$_{AH}$ [Dut] which also provides an MSO guarantees of $D^2 + 3D$. The detailed comparison of PB$_{AH}$ and our work, `FrugalSpillBound` which address similar issue is deferred to Chapter 7. Finally, we present initial directions for `OptAssist` that chooses a better of native optimizer and our robust alternatives for an input query. Note that all our techniques in thesis including that of `PlanBouquet` leverage *geometric search* techniques to achieve *deterministic* guarantees on worst-case query performance.

Adaptive

An adaptive query execution strategy, called as Lookahead Information Passing (LIP), is proposed in [ZPSP17] with the goal of being robust to cardinality estimation errors wrt join order. The key idea is to drop most of the *redundant* rows early in the joins, thus effectively avoiding a bad join order. The scope of the paper is limited to in-memory star schema and plan uses only hash join operator algorithm. Also they provide performance guarantees wrt optimal plan in the space of left-deep plans, however, they use simplistic cost model and make uniform data distribution assumption in order to achieve such guarantees.

As opposed to adaptive join ordering, an adaptive access path operator called *Smooth Scan* is introduced in [BGIA$^+$18]. This operator adapts itself to behave like index scan for low selectivity and full scan for high selectivity. Like our work, they also provide guarantees on the worst-case execution performance. However, their guarantees are a function of the underlying cost model while our guarantees are independent of it. More importantly, their technique is limited to handling estimation errors wrt only filter predicates. Interestingly, our work and these adaptive operators, including that of G-join [Gra12], can operate in conjunction wherein all the operators in plans are replaced by these adaptive operators.

2.4   Other Robustness Literature

For completeness, below we also describe robustness literature wrt non-cardinality estimation such as resource estimation, cost model errors, etc.

1. *Resource Estimation:* Estimating resources such as CPU time, memory, etc. required for query execution is also critical for choosing the optimal query execution plan. Machine learning techniques are recently used for estimation of such resources. Specifically, nearest neighbour regression is used for resource estimation in [GKD$^+$09], but it heavily relies on the input queries being very similar to training queries. In a following work [LKNC12], regression trees are used to train "scaling functions" that allow better predictions of queries not previously seen in the training data.

2. *Cost Model Estimation:* Cost model problem is important for predicting query execution time and during query planning, with cost being associated to rank competing plans. Broadly machine learning and tuning based approaches are used to attack this long standing problem. On the machine learning front, plan and operator level models are used for predicting execution time in [AÇR+12]. Further, the authors use linear regression models for each operator; that means they implicitly force the output to vary linearly with each input feature. In reality, the relationship between features and execution time is non-linear. One such relationship is captured in Chapter 7, wherein function of plan cost wrt predicate selectivities is concave.

   On the other hand, tuning the internal cost parameters of PostgreSQL engine is looked in [WCZ+13]. They do this by running a set of calibrated queries and computing the constants to be used for analytical cost model used by PostgreSQL. They achieve a Mean Relative Error (MRE) of around 40% for the TPCH bechmark queries. In the regime of such bounded cost model errors, we can show that our MSO increases by a factor of two compared to perfect cost model scenarios.

3. *Physical Database Design:* Apart from the above estimation modules, the other important factors that affect the query performance are physical and logical database designs. Recently, *CliffGuard* [MGY15] tries to come up with robust physical database design under uncertain environments such as optimizer costs or cardinality estimates. Our work is orthogonal to this aspect since we handle the uncertainty in cardinality estimation for a fixed database design.

4. *Improving the Optimization Time and Plan Quality:* There have been constant attempts on improving the optimization time while maintaining the plan quality during query compilation. Recently deep learning techniques are proposed for the problem in [KYG+18]. They improve the state-of-the-art performance with small amount of pre-training, while assuming accurate cardinality and cost model of the underlying engine. Similarly in parallel, initial approaches for the same objective again using deep learning is proposed in [MP19]. Apart from the machine learning approaches, *mixed integer linear programming* is used to handle nearly 60 tables with different join graphs such as chain, cycle and star in [TK17]. This is further improved by handling much larger scale of the problem [NR18] using search space linearization of the dynamic programming lattice.

Chapter 3

Problem Framework and Background

In this chapter, we present our query model, robustness metrics, underlying assumptions and the experimental framework. Here, we also provide a detailed description of the `PlanBouquet` technique to set up the required background for the rest of the thesis.

3.1 Selectivity Spaces

Given an SQL query q, any predicate for which the optimizer invokes the selectivity estimation module is referred to as a *selectivity predicate*, or `sp`. Consider the SPJ version of TPC-DS Query 27, which is shown in Figure 3.1. Here, each of the filter and join predicate is a `sp`. In this thesis, for simplicity, we consider `sps` to be arising out of SPJ part of the query, however, the techniques are extendable to other estimation based predicates.

For a query with D `sps`, the set of all `sps` is denoted by `SP` $= \{s_1, \ldots, s_D\}$, where s_j denotes the jth `sp`. The selectivities of the D `sps` are mapped to a D-dimensional space, with the selectivity of s_j corresponding to the jth dimension. Since the selectivity of each predicate ranges over $[0, 1]$, a D-dimensional hypercube $[0, 1]^D$ results, henceforth referred to as the *Predicate Selectivity Space* (`PSS`). Note that each location $q \in [0, 1]^D$ in the `PSS` represents a specific query instance where the `sps` happen to have the selectivities corresponding to q. Accordingly, the selectivity value of q on the jth dimension is denoted by $q.j$.

For tractability, the `PSS` is discretized at a fine-grained resolution r in each dimension. We refer to the location corresponding to the minimum selectivity in each dimension as the *origin* of the `PSS`, and the location at which the selectivity value in each dimension is maximum as the *terminus*, i.e., the other end of the principal diagonal of the `PSS`.

As shown later in Chapter 6, some selectivity dimensions may not be error-prone, and are therefore liable to be removed. Further, some other dimensions may be error-prone, but their errors do not

```
SELECT * FROM store_sales, date_dim, item,
store, customer_demographics  WHERE
ss_item_sk = i_item_sk and
ss_store_sk = s_store_sk and
ss_cdemo_sk = cd_demo_sk and
ss_sold_date_sk = d_date_sk and
cd_gender = `F' and
cd_marital_status = `D' and
cd_education_status = `Primary' and
d_year = 2000 and
s_state in (`TN')
```

Figure 3.1: TPC-DS Query 27 (SPJ version)

materially impact the overall processing cost, and can therefore also be removed. The dimensions that are retained after these pruning steps are called as *impactful error-prone predicates*, or `epp`, and they collectively form the Error-prone Selectivity Space (`ESS`).

3.2   POSP Plans

The optimal plan for a generic selectivity location $q \in$ PSS is denoted by P_q, and the set of such optimal plans over the complete PSS constitutes the *Parametric Optimal Set of Plans* (POSP) [HS02]. Note that letter subscripts for plans denote locations, whereas numeric subscripts denote identifiers. We denote the cost of executing a generic plan P at a selectivity location $q \in$ PSS by $Cost(P, q)$. Thus, $Cost(P_q, q)$ represents the optimal execution cost for the selectivity instance located at q . For ease of presentation, we will hereafter use *cost of a location* to refer to the cost of the optimal plan at that location, and denote it by $\text{COST}(q)$ ($= Cost(P_q, q)$). Finally, we assume that the query optimizer can identify the optimal query execution plan if the selectivities of all the `sps` are correctly known, for example, through the classical Dynamic Programming based search of the plan enumeration space [SAC+79].

3.3   Robustness Metrics

In our framework, the search space for robust query processing is the set of tuples $< q, P_q, \text{COST}(q) >$ corresponding to all locations $q \in$ PSS. Let us now discuss the robustness metrics we consider in our study.

3.3.1 Maximum Sub-Optimality (MSO)

We use the notion of Maximum Sub-optimality (MSO) introduced by [DH16]. The precise mathematical definition of MSO is as follows: A traditional query optimizer will first estimate $q_e \in$ PSS, and then use P_{q_e} to execute a query which may actually be located at $q_a \in$ PSS. The sub-optimality of this plan choice, relative to an oracle that magically knows the correct location, and therefore uses the ideal plan P_{q_a}, is defined as:

$$SubOpt(q_e, q_a) = \frac{Cost(P_{q_e}, q_a)}{\text{COST}(q_a)} \tag{3.1}$$

With this characterization of a specific (q_e, q_a) combination, the MSO that can potentially arise over the entire PSS is given by

$$MSO = \max_{(q_e, q_a) \in \text{PSS}} (SubOpt(q_e, q_a)) \tag{3.2}$$

As per this formulation, MSO values range over the interval $[1, \infty)$. The above definition captures the MSO of only traditional optimizers.

Let us now generalize the MSO definition to selectivity discovery algorithms like PlanBouquet. The algorithm carries out a sequence of budgeted plan executions in order to discover the location of q_a. We denote this sequence by Seq_{q_a}, with each element t_i in the sequence being a pair, (P_i, ω_i) indicating that plan P_i is executed with a maximum time budget of ω_i.

The sub-optimality of this plan sequence, denoted by $SubOpt(\text{Seq}_{q_a})$, is defined relative to an oracle that magically knows the actual or correct query location, q_a, apriori and therefore directly uses the ideal plan P_{q_a}. That is,

$$SubOpt(\text{Seq}_{q_a}) = \frac{\sum\limits_{t_i \in \text{Seq}_{q_a}} \omega_i}{\text{COST}(q_a)}$$

from which we derive

$$MSO = \max_{q_a \in \text{PSS}} SubOpt(\text{Seq}_{q_a})$$

In essence, MSO represents the *worst-case* sub-optimality that can occur with regard to plan performance over the *entire* PSS space.

3.3.2 Average Sub-Optimality (ASO)

In addition to the above primary metric, we also evaluate our proposed robust query processing algorithms over the average-case equivalent of MSO, referred by average sub-optimality (ASO). Specifically, if all q_a's are equally likely over the entire PSS, then ASO of a query processing algorithm can

be defined as follows:

$$ASO = \frac{\sum\limits_{q_a \in \texttt{PSS}} SubOpt(\texttt{Seq}_{q_a})}{\sum\limits_{q_a \in \texttt{PSS}} 1} \tag{3.3}$$

3.4   Assumptions

We make the following standard assumptions in our work which are described in detail next.

1. **Plan Cost Monotonicity (PCM)**: The notion of a location q_1 *spatially dominating* a location q_2 in the $\texttt{PSS}$ plays a central role in our robust query processing framework. Formally, given two distinct locations $q_1, q_2 \in \texttt{PSS}$, q_1 spatially dominates q_2, denoted by $q_1 \succ q_2$, if $q_1.j \geq q_2.j$ for all $j \in \{1, \ldots, D\}$. Given spatial domination, an essential assumption that allows to systematic exploration of the $\texttt{PSS}$ is that the cost functions of the plans appearing in the $\texttt{PSS}$ all obey *Plan Cost Monotonicity* (PCM). This constraint on plan cost function (PCF) behavior may be stated as follows: For any pair of distinct locations $q_b, q_c \in \texttt{PSS}$, and for any plan P,

$$q_b \succ q_c \Rightarrow Cost(P, q_b) > Cost(P, q_c)$$

   That is, it encodes the intuitive notion that when more data is processed by a query, signified by the larger selectivities for the predicates, the cost of the query processing also increases. In a nutshell, *spatial domination implies cost domination*. Apart from monotonicity, we also assume the cost functions to be continuous (smooth) throughout the $\texttt{PSS}$.

   In practice, however, we observe minor violations in monotonicity and smoothness assumptions. We overcome this issue by fitting a continuous monotonic function to the actual function. Empirically we see that this fitting can achieved with small errors.

2. **Perfect Cost Model:** Although arbitrary selectivity estimation errors are permitted in our study, we have assumed the optimizer's cost model to be perfect. While this assumption is certainly not valid in practice, improving the model quality is, in principle, an orthogonal problem to that of cardinality estimation accuracy. Furthermore, we show later that, we still can handle scenario's wherein the cost modeling errors are bounded – for instance, cost modeling error of 40% is reported in [WCHN13].

3. **Selectivity Independence:** We assume that the selectivities for the $\texttt{sps}$ are independent of each other, while this is a common assumption in much of the query optimization literature, it often does not hold in practice. Initial approaches to relax this assumption are considered in [DH16].

3.5 Database and System Framework

All our experiments in this thesis are carried out on a generic HP Z440 multi-core workstation provisioned with 32 GB RAM, 512 GB SSD and 2TB HDD. The database engine was a modified version of the PostgreSQL 9.4 engine [Pos], with the primary additions being: (a) *Selectivity Injection*, required to generate the POSP overlay of the PSS; (b) *Abstract Plan Costing*, required to cost a specific plan at a particular PSS location; (c) *Abstract Plan Execution*, required to force the execution engine to execute a particular plan; (d) *Time-limited Execution*, required to implement executions with associated time budgets; and (e) *spilling* - to execute plans in spill-mode. All the above features are explained in detail in Chapter 8. Further, the engine's configuration parameters were tuned as per PGTune [PGT].

Our test workload is comprised of a representative suite of complex OLAP queries, which are all based on queries appearing in the synthetic 100 GB TPC-DS benchmark and the 5GB real-data JOB benchmark [LGM$^+$15]. The number of relations in these queries range from 4 to 10, and a spectrum of join-graph geometries are modeled, including *chain*, *star*, *branch*, etc. Further, we have primarily modified the cyclic JOB benchmark queries to create an acyclic version of them. Since, the cyclic predicates directly nullifying our selectivity independence assumption.

We also deliberately create challenging environments for robustness by maximizing the range of cost values in the PSS. This was achieved through an index-rich physical schema that created indexes on all the attribute columns appearing in the queries.

To succinctly characterize the queries, the nomenclature aD_Qb is employed, where a specifies the number of epps, and b the query number. For example, 3D_Q15 indicates Query 15 with three of its predicates considered to be error-prone on the specified benchmark (either, TPC-DS or JOB).

3.6 Plan Bouquet Algorithm

Let us now see the PlanBouquet technique which our work builds on. As mentioned earlier, we use dimension reduction techniques to identify impactful error-prone predicates, i.e. epps, and give it as input to the PlanBouquet algorithm.

The PlanBouquet algorithm systematically discovers the actual selectivities at run-time through a sequence of cost-limited executions of a carefully chosen subset of POSP plans, called the *bouquet of plans*, or equivalently, *plan bouquet*. To understand the main ideas, we first start with the special case where the query has just *one* epp, leading to an one-dimensional ESS. Subsequently, we present the generalization to multiple dimensions.
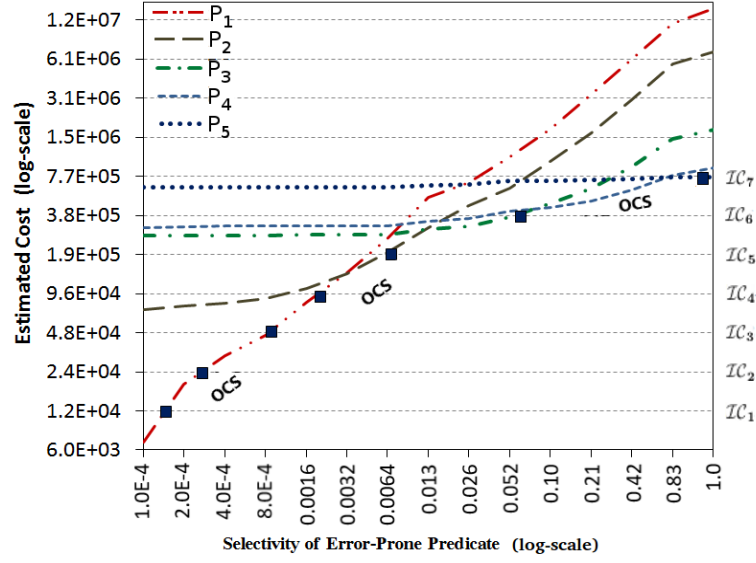
Figure 3.2: Plan Bouquet with single dimension `ESS`

3.6.1 One-dimensional `ESS`

Consider a sample 1D `ESS` shown in Figure 3.2 arising out of a query with just one `epp`. Here, the X-axis captures the selectivity range, i.e. $[0, 1]$, of the lone `epp`, while the Y-axis plots plan execution costs – note that both axes are on a log scale. There are five POSP plans, P_1, P_2, P_3, P_4, P_5, and for each of these plans, its execution costs over the `epp` selectivity range are also captured in the figure. By the definition of POSP, each plan is the best in some regime of the `ESS`. The curve corresponding to the point-wise minimum cost at each of the locations in the `ESS` is referred to as the *Optimal Cost Surface* (OCS), as indicated in the figure. We abuse notation and call OCS as a surface since it is a surface in multi-dimension, even though it is just a curve in 1D. Further, by virtue of the PCM assumption, the OCS will always be an increasing function of predicate selectivities.

Let C_{min} and C_{max} denote the minimum and maximum costs, respectively, on the OCS. Now, *discretize* the OCS by projecting a graded progression of *isocost* ($\mathcal{IC}$) steps, $\mathcal{IC}_1$ through $\mathcal{IC}_m$, onto the curve. Specifically, let the steps represent a geometric progression with common ratio 2, such that $m = \lfloor log_2(\frac{C_{max}}{C_{min}}) \rfloor + 1$ and $\mathcal{IC}_m = C_{max}$. For example, in Figure 3.2, the dotted horizontal lines represent a progression of doubling iscost steps, $\mathcal{IC}_1$ through $\mathcal{IC}_7$.

The intersection of each $\mathcal{IC}$ with the OCS (indicated by ■ in Figure 3.2), provides an associated selectivity q_i on the X-axis, along with the identity of the optimal plan P_{q_i} at this location. For example, the intersection of $\mathcal{IC}_5$ with the OCS corresponds to a selectivity of 0.0065, and associated optimal plan P_2.

26

Finally, the union of the above intersection plans forms the "plan bouquet" for the query – so, in our example, the bouquet consists of P_1, P_2, P_3, P_5.

Given the plan bouquet, which is identified at query compilation time, the 1D `PlanBouquet` algorithm operates as follows at run-time: It first picks up the bouquet plan P_{q_1}, corresponding to the smallest selectivity location q_1, and executes it with budget equal to $\mathcal{IC}_1$. If the plan is unable to complete execution within the assigned budget, the execution is aborted. Next, P_{q_2} is executed with a budget of $\mathcal{IC}_2$, followed by P_{q_3} with budget $\mathcal{IC}_3$, and so on until the budget of the executing plan is sufficient to reach completion.

Example To make the above process concrete, consider the case, with reference to Figure 3.2, where the actual selectivity of the `epp` is 0.05 , i.e. $q_a = 0.05$. To begin with plan P_1 is executed with budget equal to 1.2E4, corresponding to the cheapest isocost step $\mathcal{IC}_1$. Since the budget does not suffice (which is inferred by the non-completion), the budget is extended to $\mathcal{IC}_2$ (2.4E4) while continuing to execute the same plan. Again the budget does not suffice, and the same continues until $\mathcal{IC}_4$. Then, the plan is changed to P_2 with a budget of $\mathcal{IC}_5$ (1.9E5), but this execution too does not reach its conclusion. Finally, the execution of P_3 with budget $\mathcal{IC}_6$ (3.8E5) finishes *completely*, since the actual location, 5%, is within the selectivity range covered by $\mathcal{IC}_6$. In short, to process this query, `PlanBouquet` would invoke the execution sequence:

$$P_1|1.2E4, P_1|2.4E4, P_1|4.8E4, P_1|9.6E4, P_2|1.9E5, P_3|3.8E5$$

A salient point to note here is that whenever there is a plan switch, the results computed for the previous incomplete execution are *completely discarded* and the query evaluation begins afresh with the new plan. The important result shown, for the 1D case, was that the MSO with the above algorithm is always within **4**.

3.6.2 Multidimensional `ESS`

We now move to the general case of D-dimensional `ESS` wherein the OCS now becomes a D-dimensional surface. The optimal cost at any $q \in [0, 1]^D$ is given by $\text{COST}(q)$. Further, for a given cost C, the *isocost contour* with cost C consists of all `ESS` locations whose OCS cost is equal to C, and it will be of dimension $D - 1$.

Consider an example 2D query with two epps, resulting in an `ESS` with X and Y dimensions. The resulting 3-dimensional OCS, by incorporating a third Z dimension to capture the *cost* of the optimal plan on the `ESS`, i.e, for $q \in$ `ESS`, the value of the Z-axis corresponding to $\text{COST}(q)$, is shown in Figure 3.3. In this figure, the optimality region of each POSP plan is denoted by a unique color. So, for example, the blue region corresponds to those locations where the "blue plan" is the optimal plan.

27

Note that since Figure 3.3 is only a perspective view of the OCS, it does not capture all the POSP plans.
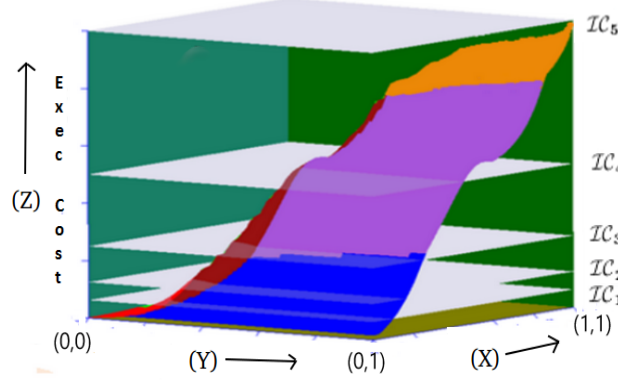
Figure 3.3: 3D Optimal Cost Surface

Discretization of OCS: As in the 1D scenario, let C_{min} and C_{max} denote the minimum and maximum costs on the OCS, corresponding to the origin and the terminus of the 3D space, respectively. We then consider $m = \lceil \log_2(\frac{C_{max}}{C_{min}}) \rceil + 1$ doubling cost hyperplanes that are parallel to the XY plane. The first hyperplane is drawn at C_{min}. For $i = 2, \ldots, m-1$, the ith hyperplane is drawn at $C_{min} \cdot 2^{i-1}$. The last hyperplane is drawn at C_{max}. These hyperplanes correspond to the m isocost contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_m$. The isocost contour $\mathcal{IC}_i$ is a one-dimensional hyperbolic curve obtained by intersecting the OCS with the ith hyperplane. For our example, as can be seen in Figure 3.4, there are 5 isocost contours for the 2D ESS. An important aspect to note here is that, unlike the 1D case, there could now be *multiple* POSP plans on an isocost contour, covering disjoint regions of the curve. For example, in Figure 3.4, with the query being located at q^* in the intermediate region between contours $\mathcal{IC}_3$ and $\mathcal{IC}_4$, there are 3 plans, P_2, P_3, P_4 on the $\mathcal{IC}_2$ contour. The set of POSP plans associated with the contour $\mathcal{IC}_i$ is denoted by PL_i. Finally, the *hypograph* of an isocost contour $\mathcal{IC}_i$ is the set of all locations $q \in \text{ESS}$ such that $\text{COST}(q) \leq \text{CC}_i$. We denote the cost of $\mathcal{IC}_i$ by CC_i.

Plan Bouquet Execution: The discovery process starts from contour $\mathcal{IC}_1$ and works its way up sequentially through the contours. When on contour $\mathcal{IC}_i$, *all* the plans in PL_i are executed with budget equal to CC_i, until one of them finishes its execution. If none of them do so, the search proceeds by jumping to the next contour $\mathcal{IC}_{i+1}$. To process the query, `PlanBouquet` would invoke the following budgeted execution sequence:

$$P_1|C, P_2|2C, P_3|2C, P_4|2C, P_5|4C, \ldots, P_{10}|4C, P_{11}|8C, P_{12}|8C$$
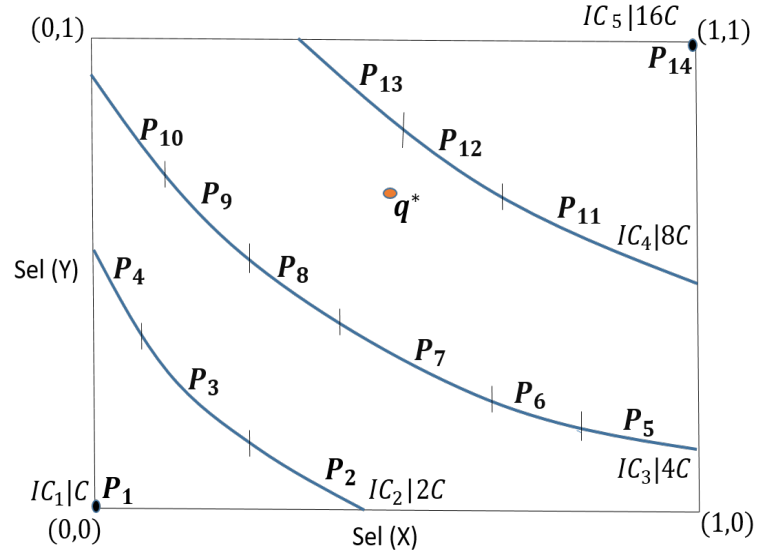
Figure 3.4: Isocost Contours in 2D ESS

with the execution of the final P_{12} plan completing the query. The MSO for this algorithm is captured in the following theorem:

Theorem 3.1 *The* `PlanBouquet` *algorithm has an MSO guarantee of 4ρ where ρ is the maximum number of plans in contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_m$, i.e., $\rho = \underset{i=\{1,\ldots,m\}}{Max} \{|\text{PL}_\mathbf{i}|\}$.*

As mentioned earlier, its bound is problematic since it depends on ρ, which is a behavioral parameter depending on a combination of the query, the optimizer, the database and the hardware platform.

For easy reference, all the notations introduced in this chapter, and predominantly used in subsequent chapters, are summarized in Table 3.1.

29

Table 3.1: NOTATIONS

Notation	Meaning
`PSS`	Predicate Selectivity Space
D	Dimensionality of the `PSS`
$s_1, ..., s_D$	Selectivity predicates in the query
`ESS`	Error-prone Selectivity Space
`epp` (`EPP`)	Error-prone predicates (its Set)
$q \in [0, 1]^D$	A location in the selectivity space
$q.j$	Selectivity of q in the jth dimension
P_q	Optimal Plan at q
q_a	Actual selectivity of query
$Cost(P, q)$	Cost of plan P at location q
COST(q)	Cost of the optimal plan P_q at location q

Chapter 4

Platform-independent Guarantees

4.1 Introduction

The `PlanBouquet` algorithm, through a calibrated sequence of cost-limited executions from a care-fully chosen set of plans, lends itself to providing an $MSO \leq 4 * \rho$, where ρ is the plan cardinality on the maximum density contour.

The `PlanBouquet` formulation, while breaking new ground, suffers from a systemic drawback – the specific value of ρ, and therefore the bound, is a function of not only the query, but also the optimizer's behavioral profile over the underlying database platform (including data contents, physical schema, hardware configuration, etc.). As a result, there are adverse consequences: (i) The bound value becomes highly variable, depending on the specifics of the current operating environment – for instance, with TPC-DS Query 25, `PlanBouquet`'s MSO guarantee of 24 under PostgreSQL shot up, under an identical computing environment, to 36 for a commercial engine, due to the change in ρ; (ii) It becomes infeasible to compute the value without substantial investments in preprocessing overheads; and (iii) Ensuring a bound that is small enough to be of practical value, is contingent on the heuristic of "anorexic reduction" [DDH07] holding true. This heuristic essentially allows POSP plans to be "swallowed" by other plans, that is, to occupy their regions in the `ESS` space, if the sub-optimality introduced due to these swallowings can be bounded to a user-defined threshold.

4.1.1 SpillBound

Our objective here is to develop a robust query processing approach that offers an MSO bound which is *solely query-dependent*, irrespective of the underlying database platform. That is, we desire a "structural bound" instead of a "behavioral bound". Accordingly, we present a new query processing algorithm, called `SpillBound`, that achieves this objective in the sense that it delivers an MSO bound that is only a function of D, the *number* of predicates in the query that are prone to selectivity

estimation errors. Moreover, the dependency is in the form of a low-order polynomial, with MSO expressed as $(D^2 + 3D)$. Consequently, the bound value becomes: (i) independent of the database platform,(ii) known upfront by merely inspecting the query, and not incurring any preprocessing overhead, (iii) indifferent to the anorexic reduction heuristic, and (iv) certifiably low in value.

SpillBound shares the core contour-wise discovery approach of PlanBouquet, but its execution strategy differs markedly. For instance, the example scenario of Figure 3.4 is again captured in Figure 4.1. The sequence of budgeted executions corresponding to SpillBound is the following (the plans are associated with tilde symbol in the figure):

$$P_1|C, P_3|2C, P_4|2C, P_7|4C, P_9|4C, P_{12}|8C$$

PlanBouquet had 12 cost budgeted executions starting with P_1 and ending with P_{12}. Note that the reduced executions with SpillBound result in cost savings of around 50% over PlanBouquet.
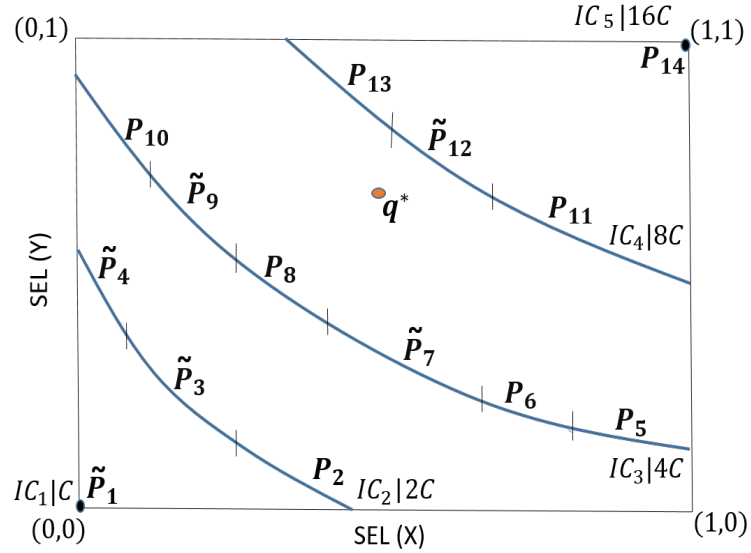
Figure 4.1: SpillBound's Execution Trace

The advantages offered by SpillBound are achieved by the following key properties – *Half-space Pruning* and *Contour Density Independent execution* – of the algorithm.

Half-space Pruning

With each contour whose plans do not complete within the assigned budget, PlanBouquet is able to prune the corresponding *hypograph* – that is, the search region *below* the contour curve. However, with SpillBound, a much stronger *half-space*-based pruning comes into play. Our half-space pruning property is achieved by leveraging the notion of *"spilling"*, whereby operator pipelines in the execution plan tree are *prematurely terminated* at chosen locations, in conjunction with *run-time*

monitoring of operator selectivities.

Contour Density Independent Execution

Let us define a "quantum progress" to be a step in which the algorithm either (a) jumps to the next contour, or (b) fully learns the selectivity of some epp (thus reducing the effective number of epps). When there are D error-prone predicates in the user query, `SpillBound` is guaranteed to make quantum progress based on cost-budgeted execution of at most D carefully chosen plans on the contour. Specifically, in each contour, for each dimension, one plan is chosen for spill-mode execution. The plan chosen for spill-mode execution is the one that provides the *maximal* guaranteed learning of the selectivity along that dimension.

Empirical Results

The summary of our experimental results indicate that for the most part, `SpillBound` provide similar guarantees to `PlanBouquet`, and occasionally, tighter bounds. As a case in point, for TPC-DS Query 26 with 4 error-prone predicates, the MSO guarantee is close to 40 with `PlanBouquet`, but comes down to 28 with `SpillBound`. With regard to empirical MSO, the `SpillBound` provides markedly superior performance over `PlanBouquet`. For instance, the empirical MSO value decreases from `PlanBouquet`'s 30.6 to 13 for `SpillBound`.

4.2 Building Blocks of our Algorithms

Let us now see in detail the two key properties of *half-space pruning* and *contour density independent execution* which forms the building blocks of `SpillBound` algorithm.

4.2.1 Half-space Pruning

`PlanBouquet` is fundamentally based on *hypograph* pruning of search spaces. By hypograph we mean the search region *below* the contour curve (after extending, if need be, the corner points of the contour to meet the axes of the search space). A pictorial view is shown in Figure 4.2, which focuses on a contour $\mathcal{IC}_i$ – here, the hypograph of $\mathcal{IC}_3$ is the region spatially below the contour.

In this work, we make a conceptual movement from hypograph pruning to a much stronger half-space pruning of the search space. Half-space pruning is the ability to prune half-spaces from the search space based on a single cost-budgeted execution of a contour plan. This is vividly highlighted in Figure 4.2, where the half-space corresponding to union of Region-1 and Region-2 is pruned by the (budget-limited) execution of P_6, while the half-space corresponding to union of Region-2 and Region-3 is pruned by the (budget-limited) execution of P_4.

We now present how half-space pruning is achieved by using *spilling* during execution of query plans. While the use of spilling to accelerate selectivity discovery had been mooted in [DH16], they
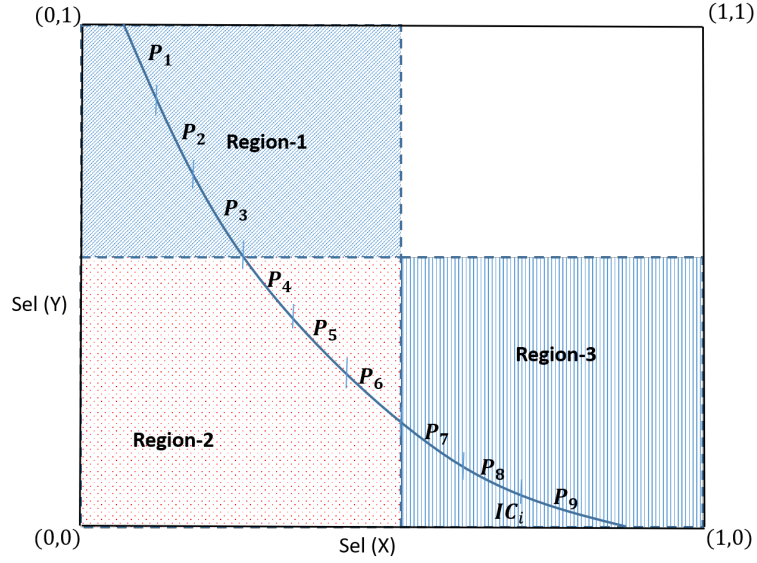
Figure 4.2: Half-Space Pruning

did not consider its exploitation for obtaining guaranteed search properties. We use spilling as the mechanism for modifying the execution of a selected plan – the objective here is to utilize the assigned execution budget to extract increased selectivity information of a specific epp. Since spilling requires modification of plan executions, we shall first describe the query execution model.

Execution Model

We assume the demand driven iterator model, commonly seen in database engines, for the execution of operators in the plan tree [Gra93]. Specifically, the execution takes place in a bottom up fashion with the base relations at the leaves of the tree.

In conventional database query processing, the execution of a query plan can be partitioned into a sequence of *pipelines* [CNR04]. Intuitively, a pipeline can be defined as the maximal concurrently executing subtree of the execution plan. The entire execution plan can therefore be viewed as an ordering on its constituent pipelines. We assume that only one pipeline is executed at a time in the database system, i.e, there is no inter-pipeline concurrency – this appears to be the case in current engines. To make these notions concrete, consider the plan tree shown in Figure 4.3 – here, the constituent pipelines are highlighted with ovals, and are executed in the sequence $\{L_1, L_2, L_3, L_4\}$.

Finally, we assume a standard plan costing model that estimates the individual costs of the internal nodes, and then aggregates the costs of all internal nodes to represent the estimated cost of the complete plan tree.

34
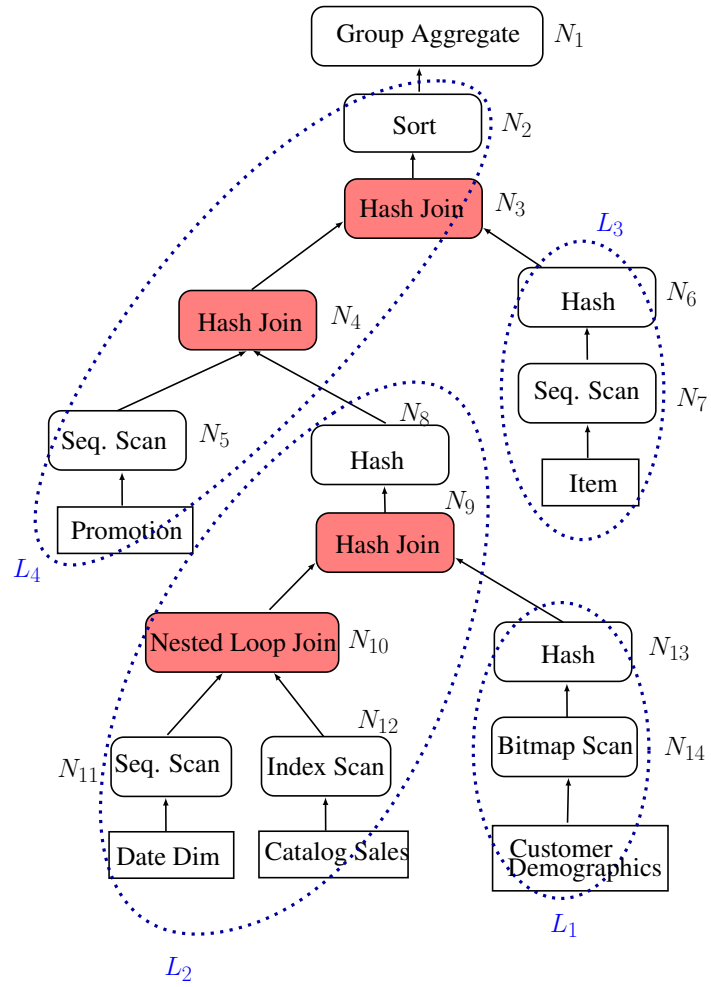
Figure 4.3: Execution Plan Tree of TPC-DS Query 26

Spill-Mode of Execution

We now discuss how to execute plans in spill-mode. For expository convenience, given an internal node of the plan tree, we refer to the set of nodes that are in the subtree rooted at the node as its *upstream* nodes, and the set of nodes on its path to the root of the complete plan tree as its *downstream* nodes.

Suppose we are interested in learning about the selectivity of an epp e_j. Let the internal node corresponding to e_j in plan P be N_j. The key observation here is that the execution cost incurred on N_j's downstream nodes in P is *not useful* for learning about N_j's selectivity. So, discarding the output of N_j without forwarding to its downstream nodes, and devoting the entire budget to the subtree rooted at N_j, helps to use the budget effectively to learn e_j's selectivity. Specifically, given plan P with cost budget B, and epp e_j chosen for spilling, the spill-mode execution of P is simply the following: Create a modified plan comprised of only the subtree of P rooted at N_j, and execute it with cost budget B.

Since a plan could consist of multiple epps (red coloured nodes in Figure 4.3), the sequence of spill node choices should be made carefully to ensure guaranteed learning on the selectivity of the chosen node – this procedure is described next.

Spill Node Identification

Given a plan and an ordering of the pipelines in the plan, we consider an ordering of epps based on the following two rules:

> *Inter-Pipeline Ordering:* Order the epps (or, EPP) as per the execution order of their respective pipelines; in Figure 4.3, since L_4 is ordered after L_2, the epp nodes N_3 and N_4 are ordered after N_9 and N_{10}.

> *Intra-Pipeline Ordering:* Order the epps by their upstream-downstream relationship, i.e., if an epp node N_a is downstream of another epp node N_b within the same pipeline, then N_a is ordered after N_b; in the example, N_3 is ordered after N_4.

It is easy to see that the above rules produce a total-ordering on the epps in a plan – in Figure 4.3, it is N_{10}, N_9, N_4, N_3. Given this ordering, we always choose to spill on the node corresponding to the *first* epp in the total-order. The selectivity of a spilled epp node is fully learnt when the corresponding execution goes to completion within its assigned budget. When this happens, we remove the epp from the set of epps and it is no longer considered as a candidate for spilling in the rest of the discovery process.

As a result of this procedure, note that the selectivities of all predicates located *upstream* of the currently spilling epp will be known *exactly* – either because they were never epps, or because they

have already been fully learnt in the ongoing discovery process. Therefore, their cost estimates are accurate, leading to the following "half-space pruning" lemma.

Lemma 4.1 *Consider a plan P for which the spill node identification mechanism identifies the predicate e_j for spilling. Further, consider a location $q \in$ ESS. When the plan P is executed with a budget $Cost(P, q)$ in spill-mode, then we either learn (a) the exact selectivity of e_j, or (b) that $q_a.j > q.j$.*

Proof: For an internal node N of a plan tree, we use $N.cost$ to refer to the execution cost of the node. Let N_j denote the internal node corresponding to e_j in plan P_q. Partition the internal nodes of P_q into the following: $Upstream(N_j)$, $\{N_j\}$, and $Residual(N_j)$, where $Upstream(N_j)$ denotes the set of internal nodes of P_q that appear before node N_j in the execution order, while $Residual(N_j)$ contains all the nodes in the plan tree excluding $Upstream(N_j)$ and $\{N_j\}$. Therefore, $\textsc{Cost}(q) = \sum_{N \in Upstream(N_j)} N.cost + N_j.cost + \sum_{N \in Residual(N_j)} N.cost$. The value of the first term in the summation is known with certainty because $Upstream(N_j)$ does not contain any epp. Further, the quantity $N_j.cost$ is computed assuming that the selectivity of N_j is $q.j$. Since the output of N_j is discarded and not passed to downstream nodes, the nodes in $Residual(N_j)$ incur zero cost. Thus, when P_q is executed in spill-mode, the budget is sufficiently large to either learn the exact selectivity of e_j (if the spill-mode execution goes to completion) or to conclude that $q_a.j$ is greater than $q.j$. □

4.2.2 Contour Density Independent Execution

We now show how the half-space pruning property can be exploited to achieve the contour density independent (CDI) execution property of the `SpillBound` algorithm. For this purpose, we employ the term "quantum progress" to refer to a step in which the algorithm either jumps to the next contour, or fully discovers the selectivity of some epp. Informally, the CDI property ensures that each quantum progress in the discovery process is achieved by expending no more than |EPP| number of plan executions.

For ease of understanding, we present here the technique for the special case of two epps referred to by X and Y, deferring the generalization for D epps to the next section.

Consider a 2D ESS shown in Figure 4.4, and assume that we are currently exploring contour $\mathcal{IC}_3$. The two plans for spill-mode execution in this contour are identified as follows: We first identify the subset of plans on the contour that spill on X using the spill node identification algorithm – these plans are identified as P_7^x, P_8^x, P_{10}^x in Figure 4.4. The next step is to enumerate the subset of locations on the contour where these X-spilling plans are optimal. From this subset, we identify the location with the *maximum* X coordinate, referred to as q_{max}^x, and its corresponding contour plan, which is denoted as P_{max}^x. The P_{max}^x plan is the one chosen to learn the selectivity of X – in Figure 4.4, this choice is P_7^x.

37
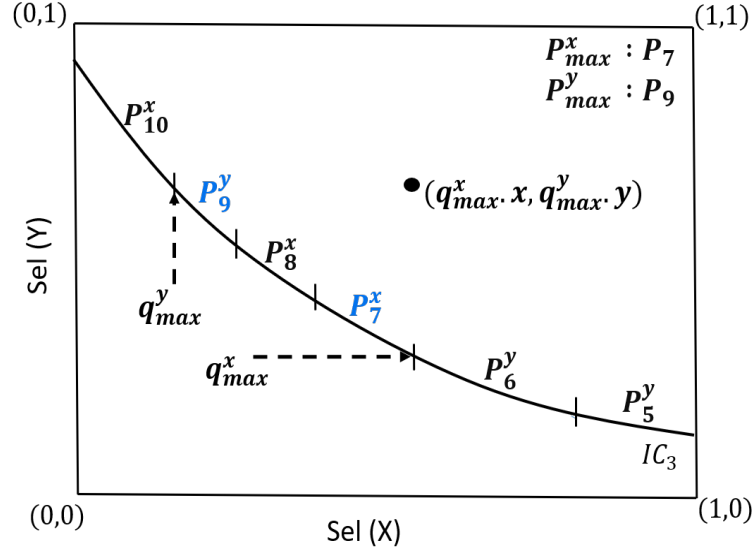
Figure 4.4: Choice of Contour Crossing Plans

By repeating the same process for the Y dimension, we identify the location q_{max}^y, and plan P_{max}^y, for learning the selectivity of Y – in Figure 4.4, the plan choice is P_9^y. Note that the location $(q_{max}^x.x, q_{max}^y.y)$ is guaranteed to be either on or beyond the $\mathcal{IC}_3$ contour.

The following lemma shows that the above plan identification procedure satisfies the CDI property.

Lemma 4.2 *In contour $\mathcal{IC}_i$, if plans P_{max}^x and P_{max}^y are executed in spill-mode, and both do not reach completion, then $\text{COST}(q_a) > \text{CC}_i$, triggering a jump to the next contour $\mathcal{IC}_{i+1}$.*

Proof: Since the executions of both P_{max}^x and P_{max}^y do not reach completion, we infer that $q_{max}^x.x < q_a.x$ and $q_{max}^y.y < q_a.y$. Therefore, q_a strictly dominates the location $(q_{max}^x.x, q_{max}^y.y)$ whose cost, by PCM, is greater than CC_i. Thus $\text{COST}(q_a) > \text{CC}_i$. □

Consider the general case of $\mathcal{IC}_i$ when there are more than two `epps`. Corresponding to an `epp` e_j, the location q_{max}^j and plan P_{max}^j are defined similar to the way q_{max}^x and P_{max}^x are defined (i.e, by replacing the X coordinate with the jth coordinate corresponding to e_j).

4.3 The SpillBound Algorithm

In this section, we present our new robust query processing algorithm, `SpillBound`, which leverages the properties of half-space pruning and CDI execution. We begin by introducing an important notation: Our search for the actual query location, q_a, begins at the origin, and with each spill-mode execution of a contour plan, we monotonically move closer towards the actual location. The running selectivity location, as progressively learnt by `SpillBound`, is denoted by q_{run}.

During the entire discovery process of `SpillBound`, only POSP plans on the isocost contour are considered for spill-mode executions. Moreover, when we mention the spill-mode execution of a particular plan on a contour, it implicitly means that the budget assigned is equal to the cost of the contour. For ease of exposition, if the epp chosen to spill on is e_j for a plan P, we shall hereafter highlight this information with the notation P^j.

For ease of exposition, we first present a version, called `2D-SpillBound`, for the special case of two epps, and then extend the algorithm to the general case of several epps.

4.3.1 `2D-SpillBound`

To provide a geometric insight into the working of `2D-SpillBound`, we will refer to the two epps, e_1 and e_2, as X and Y, respectively. `2D-SpillBound` explores the doubling isocost contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_m$, starting with the minimum cost contour $\mathcal{IC}_1$. During the exploration of a contour, two plans P^x_{max} and P^y_{max} are identified, as described in Section 4.2.2, and executed in spill-mode. The order of execution between these two plans can be chosen arbitrarily, and the selectivity information learnt through their execution is used to update the running location q_{run}. This process continues until one of the spill-mode executions reaches completion, which implies that the selectivity of the corresponding epp has been completely learnt.

Without loss of generality, assume that the learnt selectivity is X. At this stage, we know that q_a lies on the line $X = q_a.x$. Further, the discovery problem is reduced to the 1D case, which has a unique characteristic – each isocost contour of the new ESS (i.e. line $X = q_a.x$) contains only *one* plan, and this plan alone needs to be executed to cross the contour, until eventually some plan finishes its execution within the assigned budget. In this special 1D scenario, there is no operational difference between `PlanBouquet` and `2D-SpillBound`, so we simply invoke the standard `PlanBouquet` with only the Y epp, starting from the contour currently being explored.

Note that plans are *not* executed in spill-mode in this terminal 1D phase because spilling in the 1D case weakens the bound. This is because, if the plans are also executed in spilling mode in the final 1D phase, this would just lead to learning of the actual selectivity of the remaining epp. Also since the tuples are spilled (and not returned to the user), one more final execution of the optimal plan, i.e. P_{q_a} is required. Thus leading to a bound of one more than what is provided by Theorem 4.1 (also applies to multidimensional scenario).

Execution Trace

An illustration of the execution of `2D-SpillBound` on TPC-DS Query 91 with two epps is shown in Figure 4.5. In this example, the join predicate *Catalog Sales ⋈ Date Dim*, denoted by X, and the join predicate *Customer ⋈ Customer Address*, denoted by Y, are the two epps (both selectivities are
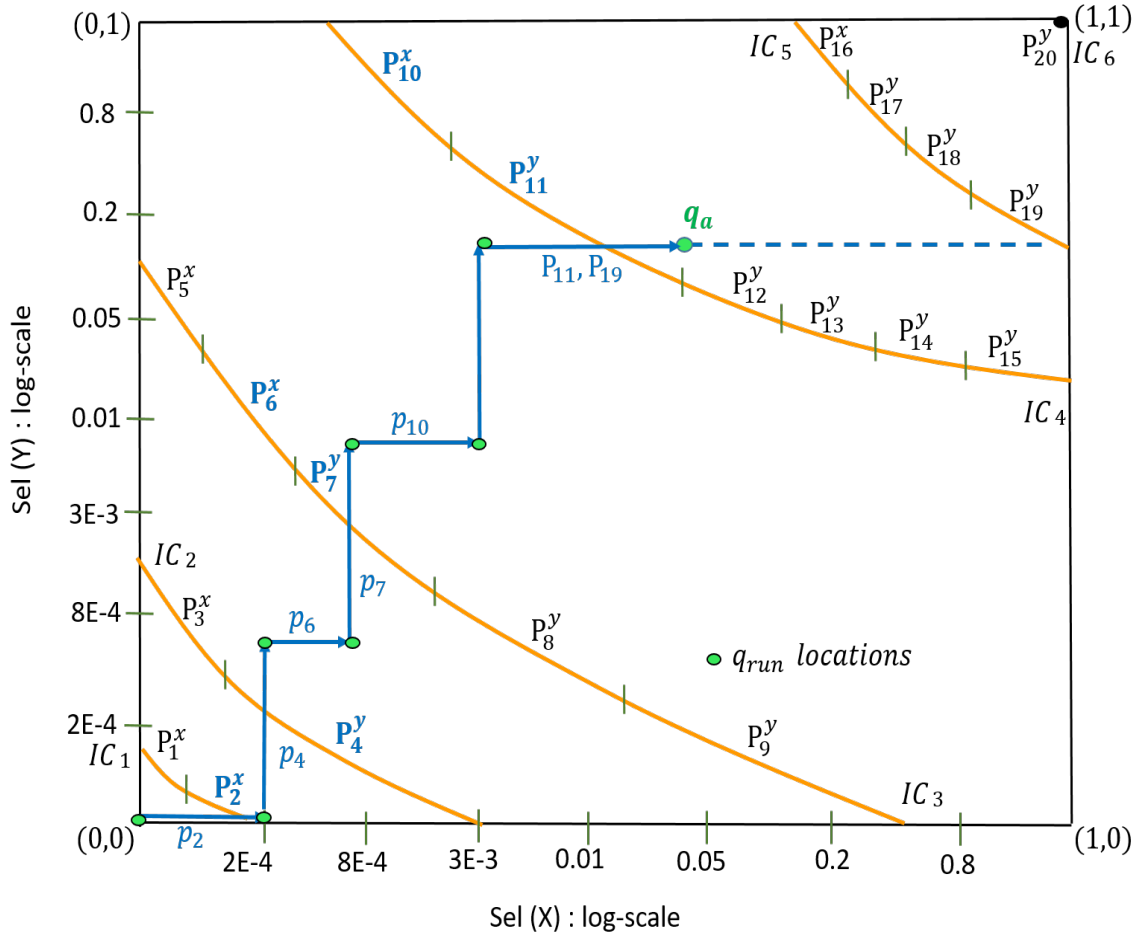
Figure 4.5: Execution trace for TPC-DS Query 91

shown on a log scale).

We observe here that there are six doubling isocost contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_6$. The execution trace of `2D-SpillBound` (blue line) corresponds to the selectivity scenario where the user's query is located at $q_a = (0.04, 0.1)$.

On each contour, the plans executed by `2D-SpillBound` in spill-mode are marked in blue – for example, on $\mathcal{IC}_2$, plan P_4 is executed in spill-mode for the epp Y. Further, upon each execution of a plan, an axis-parallel line is drawn from the previous q_{run} to the newly discovered q_{run}, leading to the Manhattan profile shown in Figure 4.5. For example, when plan P_6 is executed in spill-mode for X, the q_{run} moves from (2E-4,6E-4) to (8E-4,6E-4). To make the execution sequence unambiguously clear, the trace joining successive q_{run}s is also annotated with the plan execution responsible for the move – to highlight the spill-mode execution, we use p_i to denote the spilled execution of P_i. So, for

instance, the move from (2E-4,6E-4) to (8E-4,6E-4) is annotated with p_6.

With the above framework, it is now easy to see that the algorithm executes the sequence $p_2, p_4, p_6, p_7, p_{10}, p_{11}$, which results in the discovery of the actual selectivity of Y epp. After this, the 1D `PlanBouquet` takes over and the selectivity of X is learnt by executing P_{11} and P_{19} in regular (non-spill) mode.

This example trace of `2D-SpillBound` exemplifies how the benefits of half-space pruning and CDI execution are realized. It is important to note that `2D-SpillBound` may execute a few plans *twice* – for example, plan P_{11} – once in spill-mode (i.e., p_{11}) and once as part of the 1D `PlanBouquet` exploration phase. In fact, this notion of repeating a plan execution during the search process substantially contributes to the MSO bound in the general case of D epps.

Performance Bounds

Consider the situation where q_a is located in the region between $\mathcal{IC}_k$ and $\mathcal{IC}_{k+1}$, or is directly on $\mathcal{IC}_{k+1}$. Then, the `2D-SpillBound` algorithm explores the contours from 1 to $k + 1$ before discovering q_a. In this process,

Lemma 4.3 *The* `2D-SpillBound` *algorithm ensures that at most two plans are executed from each of the contours* $\mathcal{IC}_1, \ldots, \mathcal{IC}_{k+1}$, *except for one contour in which at most three plans are executed.*

Proof: Let the exact selectivity of one of the epps be learnt in contour $\mathcal{IC}_h$, where $1 \le h \le k + 1$. From CDI execution, we know that `2D-SpillBound` ensures that at most two plans are executed in each of the contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_h$. Subsequently, `PlanBouquet` begins operating from contour $\mathcal{IC}_h$, resulting in three plans being executed in $\mathcal{IC}_h$, and one plan each in contours $\mathcal{IC}_{h+1}$ through $\mathcal{IC}_{k+1}$. $\qquad\qquad\square$

We now analyze the worst-case cost incurred by `2D-SpillBound`. For this, we assume that the contour with three plan executions is the *costliest* contour $\mathcal{IC}_{k+1}$. Since the ratio of costs between two consecutive contours is 2, the total cost incurred by `2D-SpillBound` is bounded as follows:

$$
\begin{aligned}
TotalCost \;&\le\; 2 * \text{CC}_1 + \ldots + 2 * \text{CC}_k + 3 * \text{CC}_{k+1} \\
&=\; 2 * \text{CC}_1 + \ldots + 2 * 2^{k-1} * \text{CC}_1 + 3 * 2^k * \text{CC}_1 \\
&=\; 2 * \text{CC}_1 \left(1 + \ldots + 2^k\right) + 2^k * \text{CC}_1 \\
&=\; 2 * \text{CC}_1 \left(2^{k+1} - 1\right) + 2^k * \text{CC}_1 \\
&\le\; 2^{k+2} * \text{CC}_1 + 2^k * \text{CC}_1 \\
&=\; 5 * 2^k * \text{CC}_1 \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (4.1)
\end{aligned}
$$

From the PCM assumption, we know that the cost for an oracle algorithm (that apriori knows the

location of q_a) is lower bounded by $\texttt{CC}_{\texttt{k}}$. By definition, $\texttt{CC}_{\texttt{k}} = 2^{k-1} * \texttt{CC}_1$. Hence,

$$MSO \leq \frac{5 * 2^k * \texttt{CC}_1}{2^{k-1} * \texttt{CC}_1} = 10 \tag{4.2}$$

leading to the theorem:

Theorem 4.1 *The MSO bound of* $\texttt{2D-SpillBound}$ *for queries with two error-prone predicates is bounded by* **10**.

Remark: Note that even for a ρ value as low as 3, the MSO bound of $\texttt{2D-SpillBound}$ is better than the bound, $4 * 3 = 12$, offered by $\texttt{PlanBouquet}$.

4.3.2 Extending to Higher Dimensions

We now present $\texttt{SpillBound}$, the generalization of the $\texttt{2D-SpillBound}$ algorithm to handle D error-prone predicates $e_1, \ldots, e_D$. Before doing so, we hasten to add that the EPP set, as mentioned earlier, is constantly updated during the execution, and every epp is removed from this set as and when its selectivities become fully learnt. Further, when a contour $\mathcal{IC}_i$ is explored, the *effective search space* is the subset of locations on $\mathcal{IC}_i$ whose selectivity along the learnt dimensions matches the learnt selectivities. From now on, in the context of exploration, references to $\mathcal{IC}_i$ will mean its effective search space.

The primary generalization that needs to be achieved is to select, prior to exploration of a contour $\mathcal{IC}_i$, the best set (wrt selectivity learning) of |EPP| plans that satisfy the half-space pruning property and ensure complete coverage of the contour. To do so, we consider the (location, plan) pairs $(q_{max}^1, P_{max}^1), \ldots, (q_{max}^{|\text{EPP}|}, P_{max}^{|\text{EPP}|})$ as defined at the end of the Section 4.2.2. The set of |EPP| plans that satisfy the contour density independent execution property is $\{P_{max}^1, \ldots, P_{max}^{|\text{EPP}|}\}$.

A subtle but important point to note here is that, *during* the exploration of $\mathcal{IC}_i$, the identity of P_{max}^j *may change* as the contour processing progresses. This is because some of the plans that were assigned to spill on other epps, may switch to spilling on e_j due to their original epps being completely learnt during the ongoing exploration. Accordingly, we term the first execution of a P_{max}^j in contour $\mathcal{IC}_i$ as a *fresh execution*, and subsequent executions on the same epp as *repeat executions*.

Finally, it is possible that a specific epp may have *no* plan on $\mathcal{IC}_i$ on which it can be spilled – this situation is handled by simply skipping the epp. The complete pseudocode for $\texttt{SpillBound}$ is presented in Algorithm 1 – here, Spill-Mode-Execution($P_{max}^j, e_j, \texttt{CC}_{\texttt{i}}$) refers to the execution of plan P_{max}^j spilling on e_j with budget $\texttt{CC}_{\texttt{i}}$.

With the above construction, the following lemma can be proved in a manner analogous to that of Lemma 4.2:

Algorithm 1 The `SpillBound` Algorithm

Init: i=1, EPP = $\{e_1, \ldots, e_D\}$;
while $i \leq m$ **do**   ▷ for each contour
    if $|\text{EPP}| = 1$ **then**   ▷ only one epp left
        Run `PlanBouquet` to discover the selectivity of the remaining epp starting from the *present* contour;
        Exit;
    end if
    Run the spill node identification procedure on each plan in the contour $\mathcal{IC}_i$, and use this information to choose plan P^j_{max} for each epp e_j;
    exec-complete = false;
    for each epp e_j **do**
        exec-complete = Spill-Mode-Execution(P^j_{max},e_j,CC$_\text{i}$);
        Update $q_{run}.j$ based on selectivity learnt for e_j;
        if exec-complete **then**
            /*learnt the actual selectivity for e_j*/
            Remove e_j from the set EPP;
            Break;
        end if
    end for
    if ! exec-complete **then**
        i = i+1; /* Jump to next contour */
    end if
    Update ESS based on learnt selectivities;
end while

Lemma 4.4 *In contour $\mathcal{IC}_i$, if no plan in the set $\{P^j_{max}|e_j \in \text{EPP}\}$ reaches completion when executed in spill-mode, then* $\text{COST}(q_a) > \text{CC}_\text{i}$, *triggering a jump to the next contour* $\mathcal{IC}_{i+1}$.

Performance Bounds

We now present a proof of how the MSO bound is obtained for `SpillBound`. In the worst-case analysis of `2D-SpillBound`, the exploration cost of every intermediate contour is bounded by twice the cost of the contour. Whereas the exploration cost of the last contour (i.e., $\mathcal{IC}_{k+1}$) is bounded by three times the contour cost because of the possible execution of a third plan during the `PlanBouquet` phase. We now present how this effect is accounted for in the general case.

Repeat Executions: As explained before, the identity of plan P^j_{max} may dynamically change during the exploration of a contour $\mathcal{IC}_i$, resulting in repeat executions. If this phenomenon occurs, the new P^j_{max} plan would have to be executed to ensure compliance with Lemma 4.4. We observe that each repeat execution of an epp is preceded by an event of fully learning the selectivity of some other epp,

leading to the following lemma:

Lemma 4.5 *The* `SpillBound` *algorithm executes at most D fresh executions in each contour, and the total number of repeat executions across contours is bounded by* $\dfrac{D(D-1)}{2}$.

Proof: Consider any contour $\mathcal{IC}_i$ for $1 \leq i \leq k+1$. Note that the number of possible fresh executions on contour $\mathcal{IC}_i$ is bounded by D (in fact, it is equal to $|EPP|$ when the algorithm enters the contour $\mathcal{IC}_i$).

As mentioned earlier, a repeat execution in a contour can happen only when the exact selectivity of one of the `epps` is learnt on the contour. Let us say that when the exact selectivity of a `epp` is learnt, it marks the beginning of a new phase. If $|EPP|$ is the number of error-prone predicates just before the beginning of a phase, it is easy to see that there are at most $|EPP| - 1$ repeat executions within the phase. Further, in each phase the size of `EPP` decreases by 1. Therefore, total number of repeat executions is bounded by $\sum_{l=1}^{D-1} l = \frac{D(D-1)}{2}$. ∎ ☐

Suppose that the actual selectivity location q_a is located in the range $(\mathcal{IC}_k, \mathcal{IC}_{k+1}]$. Then, the `SpillBound` algorithm explores the contours from 1 to $k+1$ before discovering q_a. Thus, the total cost incurred by the `SpillBound` algorithm is essentially the sum of costs from fresh and repeat executions in each of the contours $\mathcal{IC}_1$ through $\mathcal{IC}_{k+1}$. Further, the worst-case cost incurred by `SpillBound` is when all the repeat executions happen at the costliest contour, $\mathcal{IC}_{k+1}$. Hence, the total cost of the `SpillBound` algorithm is given by

$$\sum_{i=1}^{k+1} (\#\text{fresh executions}(\mathcal{IC}_i)) * \text{CC}_{\text{i}} + \tfrac{D(D-1)}{2} * \text{CC}_{\text{k+1}} \tag{4.3}$$

Since the number of fresh executions on any contour is bounded by D, we obtain the following theorem:

Theorem 4.2 *The MSO bound of the* `SpillBound` *algorithm for any query with D error-prone predicates is bounded by $D^2 + 3D$.*

Proof: By substituting the values for no. of fresh executions in each contour by D in equation 4.3,

the total cost for the `SpillBound` is

$$\leq D * (\sum_{i=1}^{k+1} \texttt{CC}_\texttt{i}) + \frac{D(D-1)}{2} * \texttt{CC}_\texttt{k+1}$$

$$= D * (\sum_{i=1}^{k} \texttt{CC}_\texttt{i}) + \frac{D(D+1)}{2} * \texttt{CC}_\texttt{k+1} \qquad (4.4)$$

$$= D * (\texttt{CC}_\texttt{1} + \ldots + 2^{k-1}\texttt{CC}_\texttt{1}) + \frac{D(D+1) * 2^k \texttt{CC}_\texttt{1}}{2}$$

$$= D * (2^k - 1)\texttt{CC}_\texttt{1} + \frac{D(D+1) * 2^k \texttt{CC}_\texttt{1}}{2}$$

The cost for an oracle algorithm that apriori knows the correct location of q_a is lower bounded by $2^{k-1}\texttt{CC}_\texttt{1}$. Hence,

$$MSO \leq \frac{D * (2^k - 1)\texttt{CC}_\texttt{1} + \frac{D(D+1)*2^k \texttt{CC}_\texttt{1}}{2}}{2^{k-1}\texttt{CC}_\texttt{1}} \qquad (4.5)$$

$$\leq 2D + D(D+1) = D^2 + 3D$$

■ □

Remark 1: Note that the plan located at the end of the principal diagonal in the `ESS` hypercube is guaranteed to ensure the termination of the `2D-SpillBound` and `SpillBound` algorithms for any $q_a \in \texttt{ESS}$.

Remark 2: While proving that `PlanBouquet` delivers an MSO guarantee of $4 * \rho$, the authors of [DH16] also showed that the constant term, i.e. 4, in the guarantee is minimized when the cost ratio, cr, between the successive contours is 2. For ease of exposition of `SpillBound`, we have retained the same factor of 2. However it is interesting to note that 2 is *not* the ideal choice for `SpillBound` – in fact, the following lemma shows that `SpillBound`'s MSO is minimized by setting

$$cr = 1 + \sqrt{\frac{2}{D+1}}$$

leading to

$$MSO \leq \left(\sqrt{D} + \sqrt{\frac{D(D+1)}{2}} \right)^2$$

Lemma 4.6 *The MSO minimizing choice of cr for* `SpillBound` *is* $cr = 1 + \sqrt{\frac{2}{D+1}}$.

Proof: We know that the total cost incurred by `SpillBound` is at most $D * (\sum_{i=1}^{k+1} \texttt{CC}_\texttt{i}) + \frac{D(D-1)}{2} *$

45

$\mathtt{CC_{k+1}}$. Again considering that cost for an oracle algorithm that apriori knows the correct location of q_a is lower bounded by $cr^{k-1}\mathtt{CC_1}$, we get

$$
\begin{aligned}
MSO &\leq D * (1 + \frac{1}{cr} + \frac{1}{cr^2} + \dots) + \frac{D(D+1)}{2} * cr \\
&= D * (\frac{cr}{cr-1}) + \frac{D(D+1)}{2} * cr
\end{aligned}
\tag{4.6}
$$

$\square$ Differentiating the above MSO expression wrt cr, gives us that MSO is minimized at $cr = 1 + \sqrt{\frac{2}{D+1}}$.

So, for instance, with $D = 2$, the optimal value of cr is 1.8, resulting in an MSO of 9.9, marginally lower in comparison to the 10 obtained with $cr = 2$. Overall, for the range of D values covered in our study, only minor benefits were obtained by using the optimal cr value, and we have therefore retained the doubling factor in our evaluation.

4.4   Experimental Evaluation

The MSO guarantees delivered by `PlanBouquet` and `SpillBound` are not directly comparable, due to the inherently different nature of their dependencies on the ρ and D parameters, respectively. However, we need to assess whether the platform-independent feature of `SpillBound` is procured at the expense of a deterioration in the numerical bounds. Accordingly, we present in this section an evaluation of `SpillBound` on a representative set of complex OLAP queries, and compare its MSO performance with that of `PlanBouquet`.

The remainder of this section, for ease of exposition, we use the abbreviations `PB` and `SB` to refer to `PlanBouquet` and `SpillBound`, respectively. Further, we use MSO_g (MSO guarantee) and MSO_e (MSO empirical) to distinguish between the MSO guarantee and the empirically evaluated MSO obtained on our suite of queries.

4.4.1   SpillBound v/s PlanBouquet

The MSO guarantee for `PlanBouquet` on the original ESS typically turns out to be very high due to the large values of ρ. Therefore, as in [DH16], we conduct the experiments for `PlanBouquet` only after carrying out the *anorexic reduction* transformation [DDH07] at the default $\lambda = 0.2$ replacement threshold – we use ρ_{RED} to refer to this reduced value.

Comparison of MSO guarantees (MSO$_g$)

A summary comparison of MSO$_g$ for `PB` and `SB` over almost a dozen TPC-DS queries of varying dimensionality is shown in Figure 4.6 – for `PB`, they are computed as $4(1 + \lambda)\rho_{RED}$, whereas for `SB`, they are computed as $D^2 + 3D$.

We observe here that in a few instances, specifically 4D_Q26, 5D_Q29 and 5D_Q84, SB's guarantee is noticeably *tighter* than that of PB – for instance, the values are 28 and 38.4, respectively, for 4D_Q26. In the remaining queries, the bound quality is roughly similar between the two algorithms. Therefore, contrary to our fears, the MSO guarantee is not found to have suffered due to incorporating platform independence.
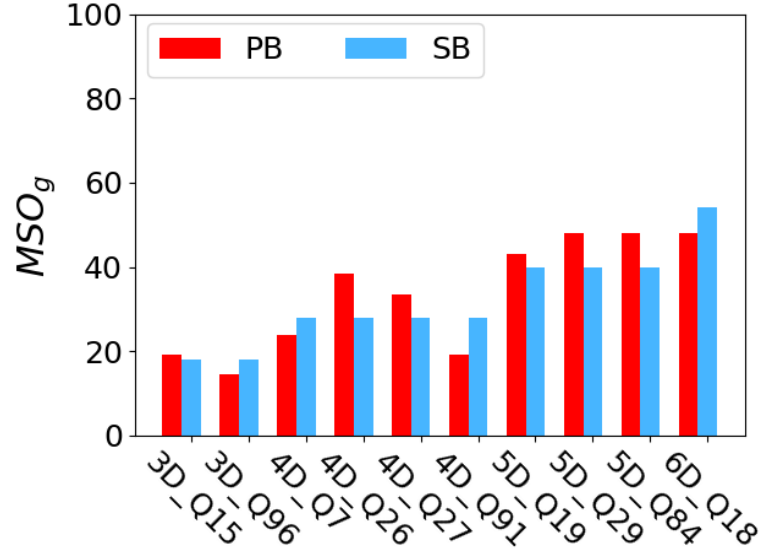
Figure 4.6: Comparison of MSO Guarantees (MSO_g)

Comparison of Empirical MSO (MSO_e)

We now turn our attention to evaluating the empirical MSO, MSO_e, incurred by the two algorithms. There are two reasons that it is important to carry out this exercise: Firstly, to evaluate the looseness of the guarantees. Secondly, to evaluate whether PB, although having weaker bounds in theory, provides better performance in practice, as compared to SB.

The assessment was accomplished by explicitly and exhaustively considering each and every location in the ESS to be q_a, and then evaluating the sub-optimality incurred for this location by PB and SB. Finally, the maximum of these values was taken to represent the MSO_e of the algorithm.

The MSO_e results are shown in Figure 4.7 for the entire suite of test queries. Our first observation is that the empirical performance of SB is far better than the corresponding guarantees in Figure 4.6. In contrast, while PB also shows improvement, it is not as dramatic. For instance, considering 6D_Q18, PB reduces its MSO from 48 to 32, whereas SB goes down from 54 to just 17.1.

The second observation is that the gap between SB and PB is *accentuated* here, with SB performing substantially better over a larger set of queries. For instance, consider query 5D_Q29, where the

MSO_g values for PB and SB were 48 and 40, respectively – the corresponding empirical values are 38 and 16.4 in Figure 4.7.

Finally, even for a query such as 4D_Q7, where PB had a marginally *better* bound (24 for PB and 28 for SB in Figure 4.6), we find that it is SB which behaves better in practice (16.4 for PB and 11.2 for SB in Figure 4.7).
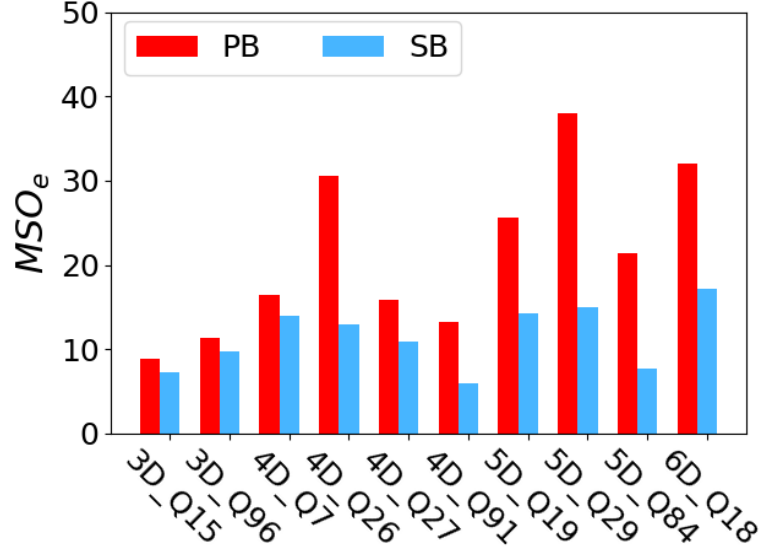
Figure 4.7: Comparison of Empirical MSO (MSO_e)

Analysis of Looseness of SB's MSO$_g$

We now profile the execution of the queries to investigate the significant gap between SB's MSO_g and MSO_e values. Recall that the analysis (Section 4.3.2) bounded the cost of repeat executions by attributing *all of them* to the last contour, i.e., $\mathcal{IC}_{k+1}$. Moreover, the number of fresh executions in all the contours, including $\mathcal{IC}_{k+1}$, was assumed to be D. This results in the execution cost over $\mathcal{IC}_{k+1}$ being the dominant contributor to MSO_g. To quantitatively assess this contribution, we present in Table 4.1 the drilled-down information of: (i) the number of fresh executions of plans on $\mathcal{IC}_{k+1}$, and (ii) the number of repeat executions of plans on $\mathcal{IC}_{k+1}$. For each of these factors, we present both the theoretical and empirical values. Note that the specific q_a locations used for obtaining these numbers corresponds to the locations where the MSO was empirically observed.

Armed with the statistics of Table 4.1, we conclude that the main reasons for the gap are the following: Firstly, while the number of repeat executions in contour $\mathcal{IC}_{k+1}$, as per the analysis, is $D(D-1)/2$, the empirical count is far fewer – in fact there is only *one* repeat execution in queries such as 3D_Q15, 4D_Q7 and 5D_Q84. While it is possible that repeat executions did occur in the

Table 4.1: SUB-OPTIMALITY CONTRIBUTION OF IC_{k+1}

Query	Fresh Executions in $\mathcal{IC}_{k+1}$		Repeat Executions in $\mathcal{IC}_{k+1}$	
	Bound	Empirical	Bound	Empirical
3D_Q15	3	2	3	1
3D_Q96	3	3	3	2
4D_Q7	4	2	6	1
4D_Q26	4	4	6	2
4D_Q27	4	4	6	3
4D_Q91	4	4	6	3
5D_Q19	5	4	10	2
5D_Q29	5	4	10	2
5D_Q84	5	3	10	1
6D_Q18	6	5	15	2

earlier lower cost contours, their collective contributions to sub-optimality are not significant.

Secondly, by the time the execution reaches the $\mathcal{IC}_{k+1}$ contour, it is likely that the selectivities of some of the epps have *already been learnt*. The bound however assumes that *all* selectivities are learnt only in the last contour. As a case in point, for 5D_Q84, the selectivities of three of the five epps had been learnt prior to reaching the last contour.

Average-case Performance (ASO)

A legitimate concern with our choice of MSO metric is that its improvements may have been purchased by degrading average-case behavior.

To investigate this possibility, we evaluated the ASO of PB and SB for all the test queries, and these results are shown in Figure 4.8. Observe that, contrary to our fears, SB provides much better performance, especially at higher dimensions, as compared to PB. For instance, with 5D_Q19, the ASO for SB is nearly 100% better than PB, going down from 17.2 to 8.8. Thus, SB offers significant benefits over PB in terms of both worst-case and average-case behavior.

Sub-Optimality Distribution

In our final analysis, we profile the *distribution* of sub-optimality over the ESS. That is, a histogram characterization of the number of locations with regard to various sub-optimality ranges. A sample histogram, corresponding to query 5D_Q84, is shown in Figure 4.9, with sub-optimality ranges of width 5. We observe here that for over 90% of the ESS locations, the sub-optimality of SB is less than 5. Whereas this performance is achieved for only 32% of the locations using PB. Similar patterns
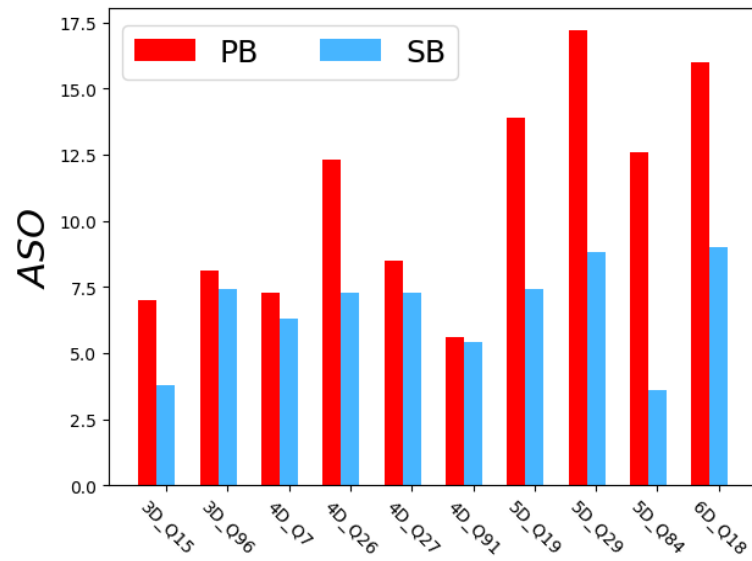
Figure 4.8: Comparison of ASO performance

were observed for the other queries as well, and these results indicate that from both *global and local* perspectives, SB has desirable performance characteristics as compared to PB.
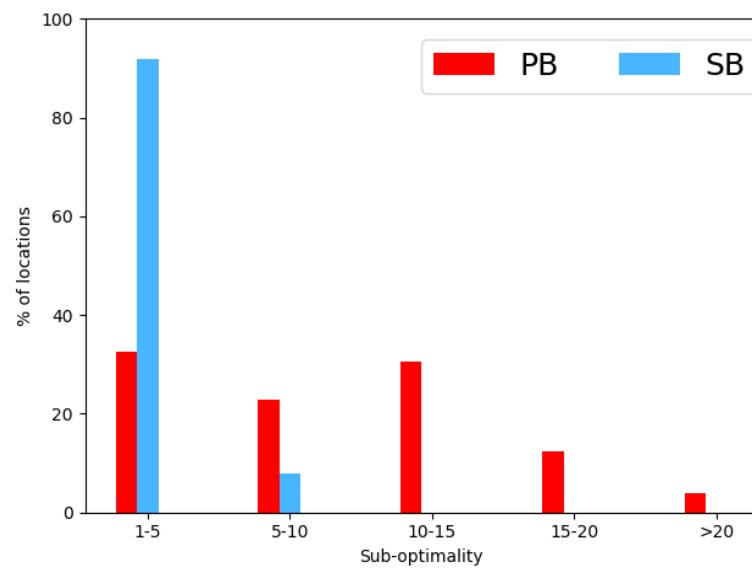
Figure 4.9: Sub-optimality Distribution (5D_Q84)

4.4.2 Wall-Clock Time Experiments

All the experiments thus far were based on optimizer cost values. We have also carried out experiments wherein the actual *query response times* were explicitly measured for the native optimizer and SB. As a representative example, we have chosen TPC-DS Q91 featuring 4 error-prone predicates, referred to as $e_1, \ldots, e_4$. In this experiment, the *optimal* plan took less than a minute (44 secs) to complete the query. However, the *native optimizer* required more than 10 minutes (628 secs) to process the data, thus incurring a sub-optimality of 14.3.

In contrast, SB took only around 4 minutes (246 secs), corresponding to a sub-optimality of 5.6. Table 4.2 shows the drilled down information of plan executions for every contour with SB. In addition, the selectivities learnt for the corresponding epp during every execution are also captured. The selectivity information learnt in each contour, shown in %, is indicated by boldfaced font in the table. Further, for each execution, the plan employed, and the overheads accumulated so far, are enumerated. A plan P executed in spill-mode is indicated with a p. As can be seen in the table, the execution sequence consists of partial executions of 13 plans spanning 6 consecutive contours, and culminates in the full execution of plan P_{10} which produces the query results.

Table 4.2: SpillBound EXECUTION ON TPC-DS QUERY 91

Contour no.	e_1 (plan)	e_2 (plan)	e_3 (plan)	e_4 (plan)	Time (sec.)
1	0	0	0	**0.08** (p_1)	1.3
2	**0.02** (p_3)	0	0	**0.3** (p_2)	7.5
3	**0.08** (p_4)	0	0	**1** (p_5)	21
4	**0.2** (p_4)	0	0	**12** (p_5)	51.2
5	**5** (p_9)	**0.8** (p_6)	0	12	86.3
5	**30** (p_9)	0.8	**5** (p_8)	**60** (p_7)	176.4
6	**80** (P_{10})	0.8	5	60	246.4

4.4.3 Evaluation on the JOB Benchmark

All the above experiments were conducted on the TPC-DS benchmark, an industry standard. Recently a new benchmark, called Join Order Benchmark (JOB), specifically designed to provide challenging workloads for current optimizers, was proposed in [LGM⁺15]. Hence, we now present results on

the primary metric, i.e. MSO, for a representative set of queries from the benchmark. The JOB results, which are enumerated in Table 4.3, are on similar lines to that of TPC-DS benchmark – thus, showcasing the platform-independent nature of `SpillBound`.

Table 4.3: RESULTS ON JOB BENCHMARK WRT MSO

Query	MSO_g		MSO_e	
	PB	SB	PB	SB
3D_Q1a	14.4	18	11.52	7.8
4D_Q13a	28.8	28	15.85	13.29
4D_Q23c	28.8	28	11.32	10.97

4.5 Conclusions

We presented `SpillBound`, a query processing algorithm that delivers a worst-case performance guarantee of $D^2 + 3D$, which is dependent solely on the dimensionality of the selectivity space. This substantive improvement over `PlanBouquet` is achieved through a potent pair of conceptual enhancements: half-space pruning of the `ESS` thanks to a spill-based execution model, and bounded number of executions for jumping from one contour to the next. Our new approach facilitates porting of the bound across database platforms, and that the bound is easy to compute as we could merely do it by query inspection. Further, it has low magnitude and is not reliant on the anorexic reduction heuristic. Further, our experimental evaluation demonstrated that `SpillBound` provides competitive guarantees to its `PlanBouquet` counterpart, while the empirical performance is significantly superior.

Chapter 5

MSO Lower Bound and its Matching Algorithm

5.1 Introduction

At this juncture, a natural question to ask is whether some alternative selectivity discovery algorithm, based on half-space pruning, can provide better MSO bounds than `SpillBound`. In this regard, we prove that *no* deterministic technique in this class can provide an MSO bound less than D. Therefore, the `SpillBound` guarantee is no worse than a factor $O(D)$ as compared to the best possible algorithm in its class.

The core idea behind the lower bound is to construct a special `ESS` and even by restricting the q_a's to be along the *perimeter* of the `ESS`, we show that atleast D half-space prunings (collectively through D executions) are required to prune the entire `ESS`.

Given this quadratic-to-linear gap on the MSO guarantee, we seek to characterize exploration scenarios in which `SpillBound`'s MSO approaches the lower bound. For this purpose, we introduce a new concept called *contour alignment*, which is described next.

Contour Alignment

A contour is aligned if the contour plan that is incident on the boundary of the `ESS`, has its selectivity learning dimension (during spill-mode execution) matching with the incident dimension. Leveraging this notion, we show that the MSO bound can be reduced to $\mathcal{O}(D)$ if the contour alignment property is satisfied at *every contour* encountered during its execution.

Unfortunately, in practice, we may not always find the alignment property satisfied at all contours. Therefore, we design the `AlignedBound` algorithm which extracts the benefit of alignment wherever available, either natively or through an explicit induction. Specifically, `AlignedBound`

delivers an MSO that is guaranteed to be in the platform-independent range $[\mathbf{2D + 2}, \mathbf{D^2 + 3D}]$.

Empirical Performance

In general, `AlignedBound`'s empirical performance is closer to the *lower end* of its guarantee range, i.e. $2D + 2$, and often provides substantial benefits for query instances that are challenging for `SpillBound`. For instance, `AlignedBound` brings the MSO of Q19 from TPC-DS benchmark down to less than 10 from `SpillBound`'s 16. In a nutshell, typically `AlignedBound` is able to complete virtually all the benchmark queries evaluated in our study with a single digit MSO.

5.2 Lower Bound on MSO

We now present a lower bound on the MSO for a class of deterministic half-space pruning algorithms denoted by $\mathcal{E}$. Consider an algorithm $\mathcal{A} \in \mathcal{E}$. Half-space pruning means the following: $\mathcal{A}$ can select an epp j and a plan P, and execute it in such a manner that the selectivity of e_j can be *partly/completely* learnt. Let $PredCost(P, e_j, \ell)$ denote the budget required by an execution of plan P, that allows $\mathcal{A}$ to conclude that $q_a.j > \ell$. For a given epp e_j, we let $CompPredCost(P, e_j)$ denote the minimum budget required by $\mathcal{A}$ to learn the selectivity of e_j completely, using P. Thus an execution of P with budget B to learn e_j allows $\mathcal{A}$ to conclude that

1. $q_a.j$ exceeds ℓ, so that $CompPredCost(P, e_j) > PredCost(P, e_j, \ell)$.

2. $q_a.j$ is at most ℓ, so that $CompPredCost(P, e_j) \leq PredCost(P, e_j, \ell)$; in this case, $q_a.j$ is learned completely.

Note that not all plans P can be used to learn e_j; in this case $PredCost(P, e_j, \ell)$ is ∞, for any $\ell \geq 0$. A spill-mode execution is one of the mechanisms for realizing half-space pruning in practice.

Given a query with an unknown selectivity q_a, the goal of $\mathcal{A}$ is to execute the query to completion. For this, the actions and outcomes of a generic step of $\mathcal{A}$ can be one of the following: (i) a plan P is executed to completion incurring $Cost(P, q_a)$, (ii) a plan P is executed with budget B and it infers that $q \not\succ q_a$ for all $q \in$ ESS with $Cost(P, q) \leq B$, and (iii) a plan P is executed with budget $PredCost(P, e_j, \ell)$, for selectivity j, and learns that (a) $q_a.j > \ell$ or (b) infer $q_a.j$ exactly.

An example of an algorithm that has the capability of executing only (i) and (ii) is `PlanBouquet`, while `SpillBound` is an example of an algorithm that has the capability of executing (i), (ii) and (iii). Thus the limitations of the algorithms in $\mathcal{E}$ apply to `PlanBouquet` and `SpillBound`. An example of an algorithm that has the capability of executing only (i) above is that of the native optimizer.

Notion of Separation: For a given $q \in$ ESS, we let $\mathcal{A}(q)$ denote the sequence of steps taken by $\mathcal{A}$, when the unknown point q_a is q. A convenient way of describing $\mathcal{A}(q_a)$, i.e. the execution of $\mathcal{A}$,

is by keeping track of the regions of the ESS where q_a is likely to be. At any step of its execution, if the action performed by $\mathcal{A}$ is hypograph pruning (action (ii)) or half space pruning (action (iii)), then it rejects certain locations in the ESS as possible q_a locations. At the completion of step t, we let $W_t^{q_a}$ be the set of locations of the ESS which are *not* pruned by $\mathcal{A}$, and let T^{q_a} be the total number of steps performed by $\mathcal{A}(q_a)$. Thus $W_0^{q_a} = \texttt{ESS}$, and we describe $\mathcal{A}(q_a)$ to be the sequence $W_0^{q_a}, W_1^{q_a}, \ldots, W_{T^{q_a}}^{q_a}$. Hence we can view the execution of $\mathcal{A}$ as a sequence of steps in which locations of the ESS are separated out from the unknown q_a, until the query is successfully executed. Note that $\mathcal{A}$ need not explicitly maintain the $W_t^{q_a}$; it is simply a means of describing the execution of $\mathcal{A}$.

We say that $\mathcal{A}(q_a)$ *separates* $q_1, q_2 \in \texttt{ESS}$ if at some step t in its execution, $q_1 \in W_t^{q_a}$, $q_2 \notin W_t^{q_a}$, while q_1, q_2 were both in $W_{t-1}^{q_a}$. More generally, for two disjoint subsets of the ESS, U_1 and U_2, we say that $\mathcal{A}(q_a)$ *separates* the set $U_1 \cup U_2$ into U_1 and U_2, if there is a step t such that $U_1 \cup U_2 \subseteq W_{t-1}^{q_a}$, but $U_1 \subseteq W_t^{q_a}$ and $U_2 \cap W_t^{q_a} = \phi$ (i.e. U_1 is a subset of $W_t^{q_a}$, while U_2 is disjoint from $W_t^{q_a}$).

Consequence of Deterministic Behavior: The algorithms we consider are deterministic. Thus the action of $\mathcal{A}$ at a step is determined completely by the actions and outcomes of previous steps. A formal way to capture this is as follows. Let q_1 and q_2 be two points of the ESS. Let t be the largest number such that $q_2 \in W_t^{q_1}$, and t' be the largest number such that $q_1 \in W_{t'}^{q_2}$. Since $W_0^{q_1} = W_0^{q_2} = \texttt{ESS}$, these points exist. At $min(t, t')$, and $W_i^{q_1} = W_i^{q_2}$ for $i = 0, 1, \ldots, t$. We are now ready to prove the lower bound.

Theorem 5.1 *For any algorithm $\mathcal{A} \in \mathcal{E}$ and $D \geq 2$, there exists a D-dimensional* ESS *where the MSO of $\mathcal{A}$ is at least D.*

Construction of ESS: Suppose the MSO of $\mathcal{A}$ is strictly less than D. We construct a special D-dimensional search space on which the contradiction is shown. It is constructed with the help of a set of locations $V = \{q_1, \ldots, q_D\}$ given by $q_i.j = 1/D$ if $j = i$, else $q_i.j = 1$. Further, our construction is such that the ESS will have exactly D plans $P_1, P_2, \ldots, P_D$. The cost structure is as follows:

$$Cost(P_i, q) = D * q.i \; \forall q \in \texttt{ESS}$$
$$Cost(P_i, q.j) = D * q.j \; \forall q \in \texttt{ESS}, \texttt{epp } j$$

Thus the POSP plan at q_i is P_i and has a cost of 1. For a two dimensional ESS and a cost c, the isocost curves correspond to L shaped objects, consisting of two segments, *blue* and *red*, as shown in the Figure 5.1. The blue segments consist of all points q with $q.x = c/2$, and $q.y \geq c/2$. Similarly, the red segments consist of all points q with $q.y = c/2$, and $q.x \geq c/2$. The points q_1 and q_2 correspond to $(1/2, 1)$ and $(1, 1/2)$ respectively.
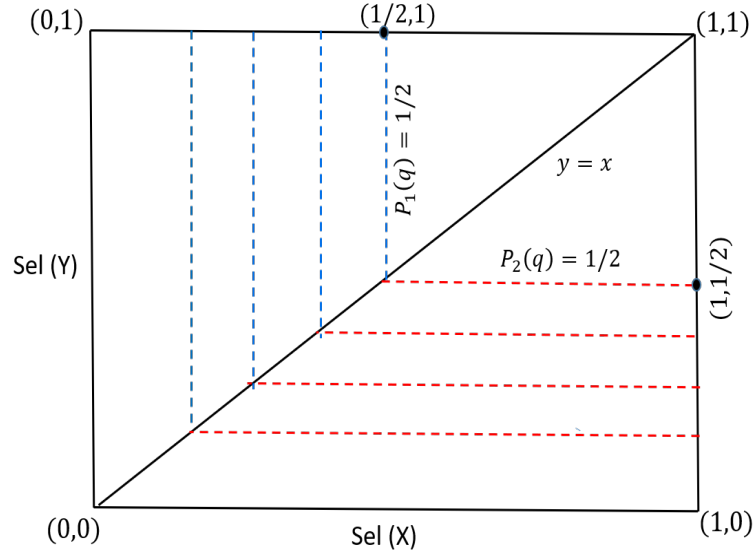
Figure 5.1: ESS for Theorem 5.1

We verify the PCM property as follows: For a plan P_j, if $q_1 \preceq q_2$, then $q_1.j \leq q_2.j$; then $Cost(P_j, q_1) = D * q_1.j \leq D * q_2.j \leq Cost(P_j, q_2)$. Note that we have allowed equality in the definition of the PCM for ease of exposition. We explain the proof with this relaxed version of the PCM, and in the last part of this section we show a modification to the costs that allows the same proof to work for the strict version of the PCM property.

Claim 5.1 *Let $q_a \in V$. Let V_1, V_2 be such that $V_1 \cap V_2 = \phi$ and $V_1 \cup V_2 = V_3 \subseteq V$. If $\mathcal{A}$ separates V_3 into V_1 and V_2, then either $|V_1| = 1$ or $|V_2| = 1$.*

If the claim is false, then $\mathcal{A}(q_a)$ splits V_3 into V_1 and V_2 each of size at least two. Let q_{i_1}, q_{i_2} and q_{i_3}, q_{i_4} be the locations in V_1 and V_2 respectively. Then $\mathcal{A}$ separates q_{i_1}, q_{i_2} from q_{i_3}, q_{i_4} in the same step. By the conditions on $\mathcal{A}$, at least one of the following must have happened.

1. $\mathcal{A}$ explores a location q and concludes that q_{i_1}, q_{i_2} both $\prec q$, while $q_{i_3}, q_{i_4} \not\prec q$ (or vice-versa, in which case interchange the roles of V_1 and V_2). By construction of V, if q_{i_1}, q_{i_2} both $\prec q$, then q has to be such that $q.j = 1 \; \forall j \in 1, \ldots, D$, i.e, $q = \mathbf{1}$. But, this implies that $q_{i_3}, q_{i_4} \preceq q$ (contradiction).

2. $\mathcal{A}$ identifies an epp j, a plan P and budget B such that $q_{i_1}.j, q_{i_2}.j$ are learned, while $q_{i_3}.j, q_{i_4}.j$ cannot be learned within budget B. Since $i_1 \neq i_2$, the budget utilized for learning the selectivities is at least D. Since $q_a \in V$, its POSP cost is 1. So, the MSO of $\mathcal{A}$ is at least D (contradicting the assumption that MSO is less than D).

56

This proves the above claim. From the above, we see that to split V_3, $\mathcal{A}$ needs a cost of at least 1. We are now ready to prove the Theorem 5.1.

Proof: *(of Theorem 5.1)* Suppose $\mathcal{A} \in \mathcal{E}$ has an MSO less than D. The POSP plan at $q_i \in V$ is P_i, and it incurs a cost of 1 to execute. The cost of executing P_i at $q_j \in V$, where $j \neq i$ is D. Since the MSO of $\mathcal{A}$ is less than D, the final step of $\mathcal{A}(q_i)$ cannot be the same for two different q_i, q_j. Thus the execution of $\mathcal{A}(q_i)$ and $\mathcal{A}(q_j)$ differs and $\mathcal{A}$ separates q_i and q_j. Choose q_a arbitrarily from $V_0 = V$ and execute $\mathcal{A}$. Consider the step in which $\mathcal{A}$ separates V_0 the first time. Suppose q_1 is separated from $V_1 = V_0 \setminus \{q_1\}$ in this step. Then choose q_a arbitrarily from V_1, and execute $\mathcal{A}(q_a)$ again. Since $\mathcal{A}$ is deterministic, $\mathcal{A}(q_1)$ and $\mathcal{A}(q_a)$ are identical till V_0 is first separated. Thus, it will first separate V_0 and then V_1. Suppose it separates q_2 from $V_2 = V_1 \setminus \{q_2\}$. Choose q_a arbitrarily from V_2, and execute $\mathcal{A}(q_a)$ again. It will first separate V_0, then V_1, and then V_2. Suppose it separates q_3 from $V_3 = V_2 \setminus \{q_3\}$. Choose q_a arbitrarily from V_3 and repeat this process inductively. Say q_D is left at the starting of Dth step, then $q_a = q_D$, $\mathcal{A}$ separates each of $V_0, V_1, \ldots, V_{D-1}$ in different steps, and finally complete q_a successfully. As each separation step needs a cost of at least 1, and a cost of at least 1 to execute q_a, $\mathcal{A}$ pays a cost of at least D for $q_a = q_D$. But, the cost of P_D at q_D is 1. Thus, the MSO of $\mathcal{A}$ is at least D, which contradicts our assumption. □

We thus have the following corollary.

Corollary 5.1 *For $D \geq 2$, there exists an* ESS, *where any deterministic half-space pruning based algorithm has an MSO of at least D*

Dealing with strict PCM: The strict PCM property is as follows: if q_1 and q_2 are two points of the ESS such that $q_1 \prec q_2$, then for all plans P, $Cost(P, q_1) < Cost(P, q_2)$. The cost function we constructed above does not satisfy this property. However, the following cost functions follow the strict PCM property. The plans are $P_1, \ldots, P_D$ as before. Their cost structure is now as follows.

$$
\begin{aligned}
Cost(P_i, q) &= D * q.i + \delta \sum_{j \neq i} q.j \ \forall q \in \text{ESS} \\
Cost(P_i, q.j) &= D * q.j \ \forall q \in \text{ESS}, \text{epp } j
\end{aligned}
$$

In the above δ is a very small positive constant whose exact value is chosen based on what we are trying to prove. Note that since the cost function is a sum of increasing linear terms, the full function is an increasing linear function.

Claim 5.2 *The above cost function $Cost()$ obey the strict PCM property.*

Proof: Let q_1 and q_2 be points in the ESS such that $q_1 \prec q_2$. Since the cost function corresponding

to any plan P_i are increasing, we have

$$D(q_1.i) + \delta \sum_{j \neq i} q_1.j \quad \leq \quad D(q_2.i) + \delta \sum_{j \neq i} q_2.j$$

So that

$$D(q_2.i - q_1.i) + \delta \sum_{j \neq i} (q_2.j - q_1.j) \quad \geq \quad 0$$

Since $q_1 \prec q_2$, the above is a sum of non-negative terms. Since the relation is strict, there is at least one k in $1, \ldots, D$, such that $q_1.k < q_2.k$, the above sum is strictly greater than zero. $\qquad \square$

Note that $Cost(P_i, q_i) = 1 + \delta(D-1)$, and $Cost(P_i, q_j) = D + \delta(D - 2 + 1/D)$. We then modify the above theorem as follows.

Theorem 5.2 *For any algorithm $\mathcal{A} \in \mathcal{E}$ and $\epsilon > 0$, for every D, there is a D-dimensional* ESS *where the MSO of $\mathcal{A}$ is at least $D - \epsilon$.*

To prove the above theorem, we note the following. Let $U \subseteq V$, and $q_i \in U$ be such that $\mathcal{A}$ separates q_i from $U \setminus \{q_i\}$. Then either $\mathcal{A}$ discovers a point q such that $q_i \preceq q$ while it does not dominate any point of U or vice versa. This means that q dominates some point of V. So the cost of executing a plan at q is at least $1 + \delta(D-1)$ which exceeds 1. Thus, to separate any two points of V, a cost of at least 1 is required.

Now suppose $\mathcal{A}$ has an MSO of at most $D - \epsilon$ for some $\epsilon > 0$. Take $\delta = \frac{\epsilon}{D^2 - 1}$. Then it is easy to verify that $Cost(P_i, q_j)/Cost(P_i, q_i)$ exceeds $D - \epsilon$. So, the final step of $\mathcal{A}(q_i)$ cannot be the same for two different q_i, q_j. The rest of the proof is on similar lines to the one of Theorem 5.1.

5.3 The `AlignedBound` **Algorithm**

Given the quadratic-to-linear gap on MSO, we now identify exploration scenarios in which the MSO of `SpillBound` matches the $\Omega(D)$ lower bound – we do so by leveraging the contour alignment notion which is described next.

5.3.1 Contour Alignment

We now introduce a key concept that helps characterize search scenarios in which the MSO of the `SpillBound` algorithm matches the lower bound. Again, for ease of understanding, we consider the special case of a 2D ESS with predicates X and Y.

Consider a contour, say $\mathcal{IC}_i$, and a dimension $j \in \{X, Y\}$. A location $q_{ext}^j \in \mathcal{IC}_i$ is said to be an *extreme location along dimension j* if the location has the maximum coordinate value for dimension

j among the contour locations belonging to $\mathcal{IC}_i$, i.e, $q^j_{ext}.j \geq q.j$, $\forall q \in \mathcal{IC}_i$. In Figure 5.2, these extreme locations are highlighted by (bold) dots.

A contour $\mathcal{IC}_i$ is said to satisfy the property of contour alignment along a dimension j if it so happens that $q^j_{max} = q^j_{ext}$, i.e., the optimal plan at q^j_{ext} spills on predicate e_j. For ease of exposition, if a contour satisfies the contour alignment property along at least one of its dimensions, then we refer to it as an *aligned contour*. In Figure 5.2, contours $\mathcal{IC}_2$ and $\mathcal{IC}_4$ are aligned along the X and Y dimensions, respectively, and are therefore aligned contours – however, contour $\mathcal{IC}_3$ is not so because it is not aligned along either dimension.
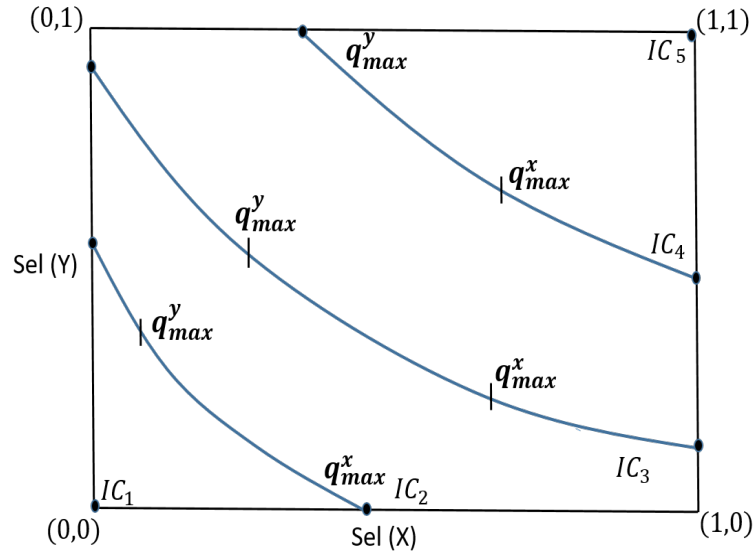
Figure 5.2: Contour Alignment

Given a contour $\mathcal{IC}_i$, Lemma 4.2 showed the sufficiency of *two* plan executions to guarantee a quantum progress in the discovery process. Leveraging the alignment notion, the following lemma describes when the same progress can be achieved with exactly *one* execution.

Lemma 5.1 *If a contour $\mathcal{IC}_i$ is aligned, then the execution of exactly one plan in spill-mode with budget* CC$_i$*, is sufficient to make quantum progress in the discovery process.*

Proof: Without loss of generality, let us assume that the contour $\mathcal{IC}_i$ satisfies contour alignment along dimension j, i.e, the optimal plan P at the location q^j_{ext} spills on dimension j. By Lemma 4.1, the spill-mode execution of P with budget CC$_i$ ensures that we either learn the exact selectivity of e_j or learn that $q_a.j > q^j_{ext}.j$. Suppose we learn that $q_a.j > q^j_{ext}.j$, then it implies that q_a lies beyond $\mathcal{IC}_i$. Thus, just the execution of P in spill-mode yields quantum progress. $\qquad\square$

Note that in the general ESS case of more than two `epps`, there may be a *multiplicity* of q_{max}^j or q_{ext}^j locations, but Lemma 5.1 can be easily generalized such that quantum progress is achieved with a single execution in these scenarios also.

5.3.2 Native Contour Alignment

Consider the scenario in which all the contours are aligned – then by Lemma 5.1, each of these contour requires only a single execution to make quantum progress. Following the lines of the analysis of `SpillBound`, and the fact that the most expensive execution sequence occurs when all the selectivities are learnt in the last contour ($\mathcal{IC}_{k+1}$), the total cost incurred in the worst-case would be:

$$
\begin{aligned}
TotalCost &= \text{CC}_1 + \cdots + \text{CC}_k + D * \text{CC}_{k+1} \\
&= \text{CC}_1 + \cdots + 2^{k-1}\text{CC}_1 + D * 2^k\text{CC}_1 \\
&\leq (2^{k-1}\text{CC}_1)(2D + 2)
\end{aligned}
$$

leading to the following theorem:

Theorem 5.3 *If the contour alignment property is satisfied at every step of the algorithm's execution, then the MSO bound is* $\mathbf{2D + 2}$.

In practice, however, the contour alignment property may not be natively satisfied at all contours – for instance, as enumerated later in Table 5.1, as few as 18 percent of the contours were aligned for a 3D ESS with TPC-DS Query 96. Therefore, we propose in this section the `AlignedBound` algorithm which operates in three steps: First, it exploits the property of alignment wherever available natively. Second, it attempts to *induce* this property, by replacing the optimal plan with an aligned substitute if the substitution does not overly degrade the performance. Finally, it investigates the possibility of leveraging alignment at a finer granularity than complete contours.

To aid in description of the algorithm, we denote by $Ext(i, j)$ the set of all extreme locations on a contour $\mathcal{IC}_i$ along a dimension j. With this, a contour $\mathcal{IC}_i$ is said to satisfy contour alignment along dimension j if $q_{max}^j \in Ext(i, j)$, i.e, at least one of the extreme locations along dimension j has an optimal plan that spills on e_j. Secondly, the set of all plans that spill on predicate e_k is denoted by $\mathcal{P}^k$.

5.3.3 Induced Contour Alignment

Given a contour $\mathcal{IC}_i$ that does not satisfy contour alignment, we *induce* contour alignment on the contour as follows: Consider a plan P which spills on $e_k \in$ EPP. It is a candidate replacement plan for any location $q_{ext}^k \in Ext(i, k)$ in order to obtain alignment along dimension k – the cost of the replacement is equal to $Cost(P, q_{ext}^k)$. Therefore, the minimum cost of inducing contour alignment

along dimension k is given by the pair $(P^k \in \mathcal{P}^k, q_{ext}^k \in Ext(i,k))$ for which $Cost(P^k, q_{ext}^k)$ is minimized. Next, we find the dimension j for which the cost of the replacement pair (P^j, q_{ext}^j) is minimum across all dimensions. Finally, the optimal plan at q_{ext}^j is replaced by P^j, and the *penalty* λ of this replacement is the ratio of $Cost(P^j, q_{ext}^j)$ to $Cost(P_{q_{ext}^j}, q_{ext}^j)$.

The usefulness of induced contour alignment depends on the penalty incurred in enforcing the property. To assess this quantitatively, we conducted an empirical study, whose results are shown in Table 5.1. Here, each row is a query instance. The "Original" column indicates the percentage of the contours that satisfy contour alignment without any replacements. A column with a particular λ value, say c, indicates the percentage of the contours satisfying contour alignment when the replacement plans are not allowed to exceed a penalty of c. The last column shows the minimum penalty that needs to be incurred for all the contours to satisfy contour alignment.

We see from the table that there are cases where full contour alignment can be induced relatively cheaply – for instance, a 50 percent penalty threshold is sufficient to make Query 5D_Q29 completely aligned. However, there also are cases, such as 3D_Q96, where extremely high penalty needs to be paid to achieve contour alignment. Therefore, we now develop a weaker notion of alignment, called *"predicate set alignment"*, which operates at a finer granularity than entire contours, and attempts to address these problematic scenarios.

Table 5.1: COST OF ENFORCING CONTOUR ALIGNMENT

Query	Original	$\lambda = 1.2$	$\lambda = 1.5$	$\lambda = 2.0$	Max λ
3D_Q96	18	18	27	45	130
4D_Q7	70	70	90	90	3.62
4D_Q26	20	30	40	50	66.95
4D_Q91	67	67	77	77	5.38
5D_Q29	40	70	100	-	1.35
5D_Q84	100	-	-	-	1

5.3.4 Predicate Set Alignment (PSA)

We say that a set $T \subseteq$ EPP satisfies predicate set alignment (PSA) with the *leader dimension* j if, for any location $q \in \mathcal{IC}_i$ whose optimal plan spills on any dimension in T, $q.j \leq q_{max}^j.j$. The set of all locations in $\mathcal{IC}_i$ whose optimal plan spills on a dimension corresponding to a predicate in T, is denoted by $\mathcal{IC}_i|T$. For convenience, we assume that the predicate corresponding to the leader dimension belongs to T. Note that PSA is a weaker notion of alignment – while contour alignment with leader dimension j mandates that $q_{max}^j.j \geq q.j$ for any $q \in \mathcal{IC}_i$, PSA only requires that $q_{max}^j.j \geq q.j$ for all $q \in \mathcal{IC}_i|T$.

Lemma 5.2 *Suppose $T_1, \ldots, T_l$ are sets of epps satisfying predicate set alignment such that $\cup_{k=1}^{k=l} T_k = \text{EPP}$, then $\cup_{k=1}^{k=l} \mathcal{IC}_i | T_k = \mathcal{IC}_i$.*

Proof: Every $q \in \mathcal{IC}_i$ spills on one of the dimensions in EPP. Therefore, it belongs to at least one $\mathcal{IC}_i | T$. □

Lemma 5.3 *Suppose $T_1, \ldots, T_l$ are sets of epps satisfying predicate set alignment such that $\cup_{k=1}^{k=l} T_k = \text{EPP}$, then spill-mode execution of l POSP plans on $\mathcal{IC}_i$ is sufficient to make quantum progress.*

Proof: Let $j_1, \ldots j_l$ be the leader dimensions for $T_1, \ldots, T_l$, respectively. Then, the l POSP plans chosen for the execution are $P_{q_{max}^{j_k}}$ for $k = 1, \ldots, l$. By definition of PSA,

$$\text{For k=1 to } l, q_{max}^{j_k}.j_k \geq q.j \quad \forall q \in \mathcal{IC}_i | T_k \tag{5.1}$$

From Lemma 5.1 and Equation 5.1, each of the $\mathcal{IC}_i | T_k$ would make quantum progress. This observation along with Lemma 5.2 proves the lemma. □

Inducing Predicate Set Alignment

Consider a contour $\mathcal{IC}_i$, and a candidate set $T \subseteq \text{EPP}$ with a leader dimension $j \in T$. We now present a mechanism to induce predicate set alignment on T with leader dimension j.

We consider the extreme location along the dimension j among all the locations in $\mathcal{IC}_i | T$, i.e, $q_T^j = \text{argmax}_{q \in \mathcal{IC}_i | T} q.j$ (in case of a multiplicity of such points, any one point can be picked). Consider the set $S = \{q \in \mathcal{IC}_i \wedge q.j = q_T^j.j\}$, i.e, all the locations belonging to $\mathcal{IC}_i$ whose coordinate value on jth dimension is equal to the coordinate value on jth dimension of an extreme location in $\mathcal{IC}_i | T$. It is easy to see that T satisfies predicate set alignment if the optimal plan at any of the locations in S is replaced with a plan P that spills on e_j. We now find a pair $(P \in \mathcal{P}^j, q \in S)$ such that $Cost(P, q)$ is minimum. The predicate set alignment property is induced by replacing the optimal plan at q with the plan P. The penalty λ for the replacement is defined as before.

We will now discuss the implementation intricacies for inducing predicate set alignment in brief. Section 4.2 explains the process of *Spill Node Identification* which produces a total ordering on the epps in a plan using *Inter-Pipeline* and *Intra-Pipeline Ordering*. Given this ordering, we choose to spill on the node corresponding to the first epp in the total-order. As a result of this procedure, the selectivities of all the predicates located in the upstream of the current spilling epp will be known exactly.

We try to achieve this property while exploring plans with a *user-defined epp*. For a query, exploration of plans inside the optimizer takes places using the dynamic programming paradigm. In the

optimizer, we change the code of generation of DP lattice such that, at each node of the DP lattice, it prunes away all plans (or sub-plans) which has *user-defined epp* in the downstream of any other epp.

Finding Minimum Cost Predicate Set Cover

Lemma 5.3 essentially says that a set of predicate sets $T_1, \ldots, T_l$ that cover EPP can be leveraged to make quantum progress. We now argue that it is sufficient to limit the search to merely the set of *partition covers* of EPP.

Consider a set T which satisfies PSA along dimension j. The *cover cost* of $T_1, \ldots, T_l$ is said to be sum of cost of enforcing PSA for each of the T_is. We say that T satisfies *maximal* PSA with leader dimension j if no super-set of T satisfies the property with same or lesser cost. Consider $T_1, \ldots, T_l$ which cover EPP and have been enforced to satisfy maximal PSA. We now obtain a partition cover whose cover cost is at most the cover cost of $T_1, \ldots, T_l$.

Let $j_1, \ldots, j_l$ be the leader dimensions for $T_1, \ldots, T_l$. The maximal property of the T_is implies that no dimension can be a leader dimension for more than one T_i. Therefore, the following sets $\pi_1 = T_1 + \{j_1\} - \cup_{m=2}^{m=l}\{j_m\}, \pi_k = T_k + \{j_k\} - \cup_{m=1,m\neq k}^{m=l}\{j_m\} - \cup_{m=1}^{m<k}\pi_m$ for $k = 2, \ldots l-1$, and $\pi_l = T_l - \cup_{m=1}^{m=l-1}\pi_m$ provide a partition cover with the same set of leader dimensions $j_1, \ldots, j_l$. It follows that the cover cost of $\pi_1, \ldots, \pi_l$ is at most the cover cost of $T_1, \ldots, T_l$.

Let $j_1, \ldots j_l$ be the leader dimensions for $T_1, \ldots, T_l$. Let $C_{PSA}(T_i, j_i)$ denote the minimum cost required to induce Predicate Set Alignment for set T_i with leader dimension j_i. The maximal property of the T_is implies that no dimension can be a leader dimension for more than one T_i. Therefore, the following sets $\pi_1 = T_1 + \{j_1\} - \cup_{m=2}^{m=l}j_m, \pi_k = T_k + \{j_k\} - \cup_{m=1}^{m=l,m\neq k}\{j_m\} - \cup_{m=1}^{m<k}\pi_m$ for $k = 2, \ldots l-1$, and $\pi_l = T_l - \cup_{m=1}^{m=l-1}\pi_m$ provide a partition cover with the same set of leader dimensions $j_1, \ldots, j_l$.

Lemma 5.4 $C_{PSA}(T/j) \geq C_{PSA}(T'/j)$, *if* $T' \subseteq T$

Proof: By definition of leader dimension, if j induces Predicate Set Alignment on T with cost $C_{PSA}(T/j)$, it also induces Predicate Set Alignment on T' with atmost the same cost. $\square$

Lemma 5.5 $\sum_{i=1}^{i=l} C_{PSA}(T_i/j_i) \geq \sum_{i=1}^{i=l} C_{PSA}(\pi_i/j_i)$

Proof: From above, we know that no dimension can be leader dimension for more than one T_i. Since we are removing dimensions from T_i to obtain π_i, $\pi_i \subseteq T_i$. Moreover, the leader dimension of T_i and π_i is same. Thus, from Lemma 5.4 and the above arguments, we prove the lemma. $\square$

Thus the cover cost of $\pi_1, \ldots, \pi_l$ is at most the cover cost of $T_1, \ldots, T_l$. Therefore, we can restrict the search for EPP cover to only partition covers without incurring any increase in the penalty of the

EPP cover. The benefit of this is that the number of partition covers of a set is much smaller than the number of different ways of covering a set with its subsets.

Given a partition cover $\pi = \{\pi_1, \ldots, \pi_l\}$, π_λ denotes the sum of the penalties incurred in enforcing PSA for each of the π_is along their leader dimensions.

5.3.5 Algorithm Description

The `AlignedBound` algorithm is presented in Algorithm 2. The steps that are identical to the steps in `SpillBound` are not presented again and simply captured as comments.

The key steps of the algorithm are **S1** and **S2** which are executed using the partition cover and predicate set alignment techniques are described in Section 5.3.4.

A legitimate concern at this point is whether in trying to induce alignment, the $D^2 + 3D$ guarantee may have been lost along the way. The key to the analysis is an alternate way of understanding the $O(D^2)$ MSO of `SpillBound`. At each inner for-loop of `SpillBound` it incurs a penalty of |EPP|, i.e, a penalty of 1 for each of the epp in EPP. On the last contour, in the outer while-loop, the penalty of the inner for-loop is incurred for at most $D - 1$ repeat executions.

With this perspective, we prove the following theorem.

Theorem 5.4 *The MSO bound of* `AlignedBound` *algorithm for any query with D error-prone predicates is bounded by* $D^2 + 3D$.

Proof: (i) At each execution of **S1** step, there is a trivial way to obtain penalty equal to |EPP| by considering just singleton parts corresponding to each remaining epp. So, the penalty of this step is upper bounded by |EPP|, (ii) The number of repeat executions also continues to be bounded by $D - 1$ as in the case of `SpillBound`. So MSO is bounded by $D^2 + 3D$, similar to `SpillBound`. □

Theorem 5.5 *In the best case, MSO bound of* `AlignedBound` *is* $O(D)$.

Proof: In the best case, the penalty of the chosen partitions in the **S1** steps is a constant. This can happen even when contour alignment is not satisfied, because a partition cover with constant number of parts, each having a constant penalty, is also sufficient to obtain a constant penalty at step **S1**. This will lead to MSO of $O(D)$ □

Thus, it captures a larger set of search scenarios in which an MSO of $O(D)$ can be obtained. Finally, from an empirical point of view, the algorithm is designed to take advantage of PSA to whatever extent possible during the search. We show the empirical benefits of this optimization in the experimental section, especially for query instances on which the empirical MSO of `SpillBound` is relatively larger.

Algorithm 2 The `AlignedBound` Algorithm
___
 1: **Init:** i=1, EPP=$\{e_1, \ldots, e_D\}$;
 2: **while** $i \leq m$ **do**   ▷   for each contour
 3:     /* Handle special 1-D case when it is encountered */
 4:     **S0:** Π = Set of all partitions of EPP (remaining `epps`);
 5:     **S1:** We pick $\pi \in \Pi$ with minimum π_λ;
 6:     **for each** part $\pi_k \in \pi$ **do**
 7:         **S2:** Let j_k be the leader dimension, P the replacement plan along dimension j_k, and q the location whose optimal plan is replaced with P;
 8:         exec-complete = Spill-Mode-Execution($P, e_{j_k}, Cost(P, q)$);
 9:         Update $q_{run}.j_k$ based on selectivity learnt for e_{j_k};
10:         **if** exec-complete **then**
11:             Remove e_{j_k} from the part π_k and the set EPP;
12:             Break;
13:         **end if**
14:     **end for**
15:     /* Update `ESS`, jump contour as in `SpillBound` */
16: **end while**
___

5.4   Experimental Evaluation

Let us now assess the empirical performance of `AlignedBound` over `SpillBound`. In addition to modifications to engine such as spilling mentioned before, we implement a feature that obtains a least cost plan from optimizer which spills on a user-specified `epp`. This is primarily needed for `AlignedBound` algorithm to find the minimum penalty replacement pair which is mentioned in Section 5.3.

The remainder of this section, for ease of exposition, we use the abbreviations `SB` and `AB` to refer to `SpillBound` and `AlignedBound`, respectively. Further, as mentioned before, we use MSO_g (MSO guarantee) and MSO_e (MSO empirical) to distinguish between the MSO guarantee and the empirically evaluated MSO obtained on our suite of queries.

5.4.1   Comparison of Empirical MSO

We now see how the predicate set alignment (PSA) property, exploited by `AB`, impacts its empirical performance as compared to `SB`. Specifically, we assess the MSO_e incurred by the two algorithms, along with the comparison on other metrics, such as ASO and sub-optimality distribution.

The MSO_e numbers for `SB` and `AB` are captured in Figure 5.3. First, we highlight that the MSO_e values for `AB` are consistently less than around 10, for all the queries. Second, `AB` significantly brings down the MSO_e numbers for the queries whose MSO_e values with `SB` are greater than 15. As a case

in point, `AB` brings down the MSO_e of 6D_Q18 from 17.1 to 10.8.
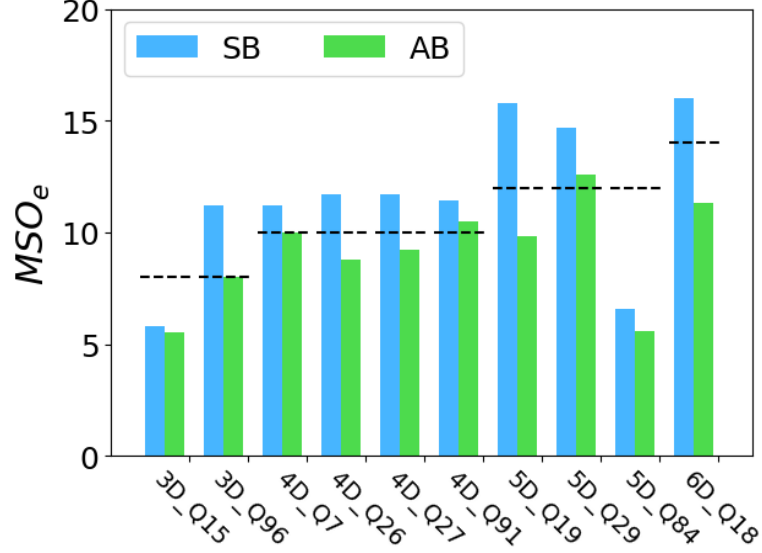
Figure 5.3: Comparison of Empirical MSO (MSO_e)

Rationale for `AB`'s Performance Benefits

Recall that `AB` provides an MSO guarantee in the range $[2D + 2, D^2 + 3D]$. As can be seen in Figure 5.3, the MSO_e values for `AB` are closer to the corresponding $2D + 2$ bound value, shown with dotted lines in the figure. These results suggest that the empirical performance of `AB` is close to the $\mathcal{O}(D)$ lower bound on MSO.

We now shift our focus to examining the reasons for `AB`'s MSO_e performance benefits over `SB`. In Table 5.2, the maximum penalty over all partitions encountered during execution is tabulated for the various queries. The important point to note here is that these penalty values are lower than 3, even for the 6D query. Since the highest cost investment for quantum progress in any contour is the maximum penalty times the cost of the contour, the low value for penalty results in the observed benefits, especially for higher dimensional queries.

5.4.2 Comparison of ASO

Moving our attention to average case metric of MSO, we see in Figure 5.4 that the `AB`'s ASO numbers improve significantly over `SB`. As a case in point, for 6D_Q18 the ASO reduces from 9.8 for `SB` to 4.7 for `AB`.

Table 5.2: MAXIMUM PENALTY FOR AB

Query	Max. Penalty for AB
3D_Q15	2.0
3D_Q96	3
4D_Q7	3
4D_Q26	2.7
4D_Q27	2.9
4D_Q91	2
5D_Q19	3
5D_Q29	1.8
5D_Q84	1.1
6D_Q18	1.8
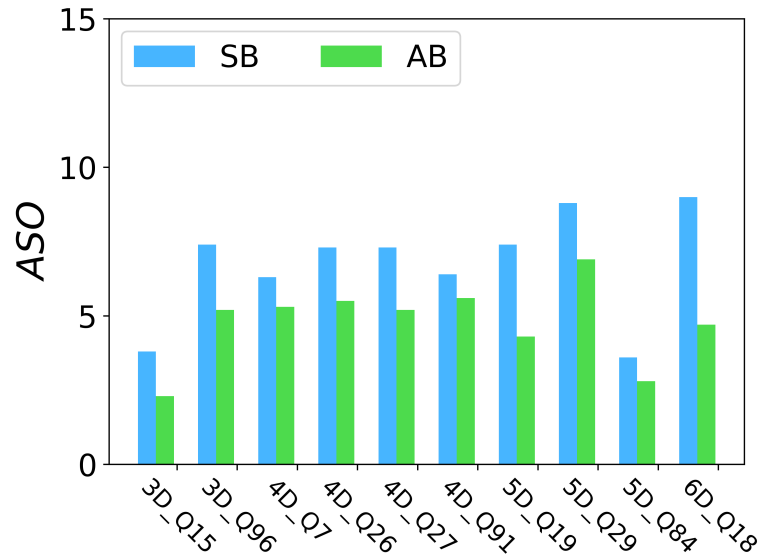
Figure 5.4: Comparison of ASO performance

5.4.3 SubOptimality Distribution

We now profile the distribution of the sub-optimalities over the ESS. In Figure 5.5, we observe that nearly 80% of the `ESS` locations have sub-optimality less than 6 when we use `AlignedBound` algorithm. This is much more as compared to `SpillBound` which only has around 43% locations within this performance range. Similar pattern was observed for other queries as well. These results indicate that `AlignedBound` has desirable performance characteristics as compared to `SpillBound`.
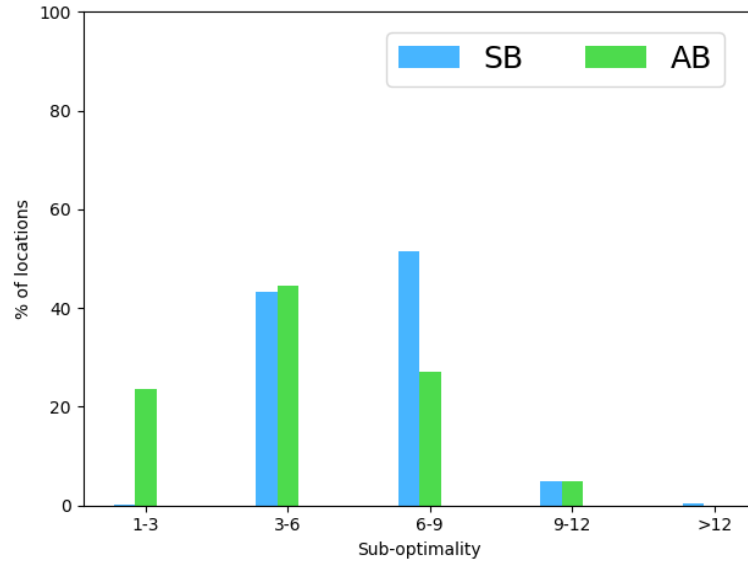
Figure 5.5: Sub-optimality distribution (5D_Q19)

5.4.4 Evaluation on the JOB Benchmark

We now present results, in Table 5.3, for the same representative queries from the JOB benchmark as captured in Table 4.3. Here as well we see that the MSO_e values for `AB` are less than around 10 for all the queries, along with performing better than `SB`.

Table 5.3: RESULTS ON JOB BENCHMARK WRT MSO

Query	MSO_e	
	SB	AB
3D_Q1a	7.8	6.4
4D_Q13a	13.29	9.12
4D_Q23c	10.97	7.24

5.5 Conclusions

In this chapter, we first proved a lower bound of D on MSO among the class of half-space pruning algorithms. Then, we introduced the contour alignment and predicate set alignment properties, and leveraged them to design `AlignedBound` with the objective of bridging the quadratic-to-linear MSO gap between `SpillBound` and the lower bound. Our detailed empirical evaluation suggests that `AlignedBound`'s empirical performance often approaches the ideal of MSO linearity in D.

Chapter 6

Dimensionality Reduction

6.1   Introduction

Notwithstanding the unique and welcome benefits of the `SpillBound` class of techniques with regard to robust query processing, it suffers from the *"curse of dimensionality"* on two important fronts – firstly, the overheads of constructing the POSP overlay are *exponential* in the `PSS` dimensionality, and secondly, the MSO guarantees are *quadratic* in this dimensionality.

For ease of presentation, we use `SpillBound` as a representative algorithm for `AlignedBound` unless otherwise mentioned. Just to recall that the MSO of `SpillBound` has the following upper bound:

$$MSO_{SB}(D) \leq D^2 + 3D \tag{6.1}$$

Contemporary OLAP queries often have a high ab initio `PSS` dimensionality, a legitimate question that arises is whether the `SpillBound` technique can be made practical for current database environments. As a case in point, consider the SPJ version of TPC-DS Query 27 shown in Figure 6.1, whose raw dimensionality is **9** (comprised of 4 join predicates, 4 equality filter predicates, and 1 set membership predicate). Constructing its `PSS` at even a modest resolution of $r = 20$ (corresponding to 5% increments in the selectivity space) would require making about 0.5 *trillion* calls to the query optimizer, and the MSO would exceed *100*.

Earlier, the `PSS` dimensionality issue was handled by manually identifying and eliminating dimensions that were either accurately estimated by the optimizer, or whose errors did not materially impact the overall plan performance – the resulting reduced space was termed as the Error Selectivity Space (`ESS`). This inspection-based approach is not a scalable solution, and moreover, may have missed opportunities for dimension removal due to its ad-hoc nature. We therefore propose an automated technique, called `DimRed`, for converting high-dimensional `PSS` into equivalent low-dimensional

```
SELECT * FROM store_sales, date_dim, item,
store, customer_demographics  WHERE
ss_item_sk = i_item_sk and
ss_store_sk = s_store_sk and
ss_cdemo_sk = cd_demo_sk and          ⎤  WeakDim
ss_sold_date_sk = d_date_sk and       ⎦  Removal
cd_gender = `F' and                   ⎤
cd_marital_status = `D' and              MaxSel
cd_education_status = `Primary'  and  ⎦  Removal
d_year = 2000 and                     ⎤  Schematic
s_state in (`TN')                     ⎦  Removal
```
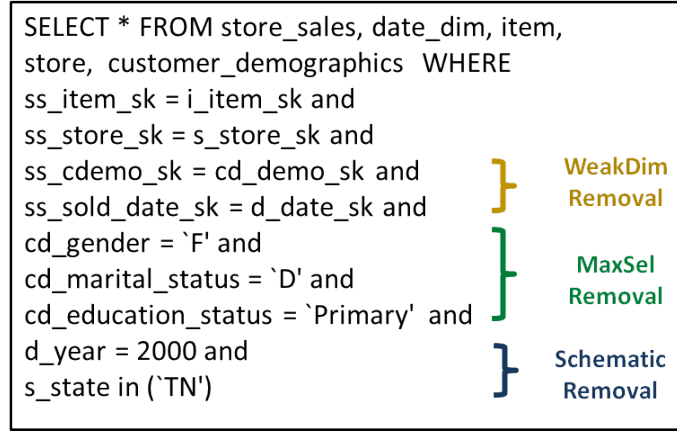
Figure 6.1: TPC-DS Query 27 (SPJ version)

ESS, and report on the outcomes here.

6.2 Problem Definition

Given the above framework, a mandatory criterion for our dimensionality reduction algorithm is that the MSO of the resultant ESS should be no worse than that of the initial PSS – that is, the reduction should be *"MSO-safe"*. Within this constraint, there are two ways in which the optimization problem can be framed – we can choose to either minimize the compilation overheads, or to minimize the MSO, leading to the following problem definitions:

Overheads Metric: Develop an MSO-safe time-efficient ESS construction algorithm that, given a query Q with its PSS, removes the maximum number of PSS dimensions.

MSO Metric: Develop an MSO-safe time-efficient ESS construction algorithm that, given a query Q with its PSS, removes a set of PSS dimensions such that the resulting MSO is minimized.

6.3 Outline of the DimRed Procedure

We now present an outline of DimRed which incorporates a pipeline of reduction strategies whose collective benefits ultimately result in ESS dimensionalities that can be efficiently handled by modern computing environments.

The DimRed procedure is composed of three components: SchematicRemoval, MaxSelRemoval, and WeakDimRemoval that are applied in sequence in the processing chain from the user query submission to its execution with SpillBound, as shown in Figure 6.2. Here, SchematicRemoval and MaxSelRemoval reduce the overheads through *explicit* removal of

71

PSS dimensions, whereas `WeakDimRemoval` improves the MSO through *implicit* removal of dimensions. To illustrate `DimRed`'s operation, we use TPC-DS Query 27 (Figure 6.1) as the running example.
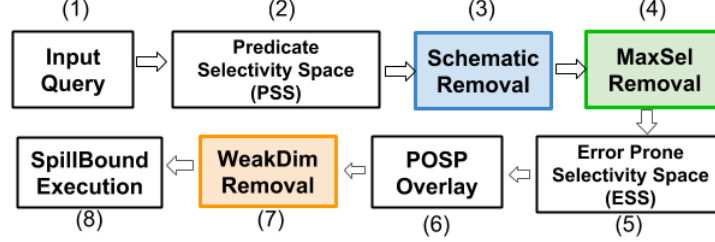
Figure 6.2: `DimRed` Pipeline

In the first component, `SchematicRemoval`, a dimension d is removed whenever we expect that the selectivity estimates made by the optimizer, using either the metadata statistics or the physical schema, will be highly accurate. For instance, database engines typically maintain exact frequency counts for columns with only a limited number of distinct values in their domains – therefore, the dimensions corresponding to the d_year and s_state columns in Q27 can be safely removed from the PSS.

The second component, `MaxSelRemoval`, takes a cost-based approach to identify "don't-care" dimensions where the actual selectivity value does not play a perceptible role on overall performance. Specifically, given a candidate dimension d, it conservatively assumes that dimension d's selectivity is the *maximum* possible (typically, 1). Due to this movement to the ceiling value, there are countervailing effects on the MSO guarantee – on one hand, the value of D is decremented by one in Equation 6.1, but on the other, an *inflation factor* α_d is suffered due to `SpillBound`'s choice of bouquet plan executions being now dictated by the maximum selectivity, rather than the actual value. The good news is that α_d can be bounded and efficiently computed, as explained in Section 6.5. Therefore, we can easily determine whether the benefits outweigh the losses, and accordingly decide whether or not to remove a dimension. For instance, in Figure 6.1, the three filter predicates on cd_gender, cd_marital_status and cd_education_status can be removed since their inflation factors are small, collectively amounting to just 1.34.

After the explicit dimension removal by the `SchematicRemoval` and `MaxSelRemoval` components, the POSP overlay of the `ESS` is computed. Subsequently, rather than separately discovering the selectivity of each of the remaining dimensions, we attempt, using the `WeakDimRemoval` component, to *piggyback* the selectivity discovery of relatively "weak" dimensions on their "strong" siblings – here, the strength of a dimension is characterized by its α_d value, as computed previously by the `MaxSelRemoval` module. That is, a dimension d with low α_d is discovered concurrently

72

with a high inflation counterpart – again, there are countervailing factors since the concurrent discovery effectively reduces the dimensionality by one, but incurs a second inflation factor β_d due to the increased budgetary effort incurred in this process. However, the good news again is that β_d can be bounded and efficiently computed if the ESS is available, as explained in Section 6.6, and it can be easily determined whether the benefits outweigh the losses. For instance, in Figure 6.1, the last two join predicates are implicitly removed through this process, since their execution is piggybacked on the first two join predicates.

Performance Results

The summary theoretical characterization of the DimRed procedure, with regard to dimensionality, computational overheads and MSO guarantees, is captured in Table 6.1. In this table, k_s and k_m denotes the number of dimensions that are explicitly removed thanks to the SchematicRemoval and MaxSelRemoval components, respectively, while k_w denotes the number of implicitly removed dimensions from the WeakDimRemoval component. α_M captures the collective MSO inflation factor arising from the removal of the k_m don't-care dimensions, while β_W indicates the net inflation factor arising out of the k_w piggybacked discoveries. The last column compares the cumulative overheads, captured by r^D, incurred after applying the DimRed pipeline relative to what would have been incurred on the native PSS.

Note that there is a optimized variant, called Nexus proposed in [DH16], which only discover parts of the PSS corresponding only to the contours. Even though there are material reductions in compilation overheads but it still has exponential dependency on the dimensions. For reasons detailed in Section 7.7.1, we use complete exploration of PSS, requiring r^D optimizer calls for compilation overheads.

Table 6.1: Summary Performance Characterization

	Dimensionality	**Maximum Suboptimality**	**Overheads (Opt Calls)**
PSS	D	$MSO_{SB}(D)$	r^D
Schematic Removal	$D - k_s$	$MSO_{SB}(D - k_s)$	$r^{D-k_s-k_m}+$ $\theta(2^D)$
MaxSel Removal	$(D - k_s - k_m)$	$\alpha_M *$ $MSO_{SB}(D - k_s - k_m)$	
WeakDim Removal	$(D - k_s - k_m - k_w)$	$\alpha_M * \beta_W *$ $MSO_{SB}(D - k_s - k_m - k_w)$	

Given this characterization, we need to assess whether the values of k_s, k_m and k_p are substantial enough in practice to result in a low-dimensional ESS, and we have therefore conducted a detailed empirical evaluation of the DimRed procedure. Specifically, we have evaluated its behavior on a

representative suite of 50-plus queries, sourced from the popular TPC-DS and JOB benchmarks. Our results indicate that `DimRed` is consistently able to bring down the `PSS` dimensionality of the workload queries, some of which are as high as 19, to 5 or less. A sample outcome for Query 27 is shown in Table 6.2, and we see here that the original dimensionality of 9 is brought down to as low as 1 when optimizing for overheads, and as low as 2 when optimizing for MSO. Further, the preprocessing time taken before query execution can actually begin is now down to seconds from days. Finally, the resulting MSOs are not only safe, but significantly better than those on the original system – for instance, optimizing for MSO produces a huge improvement from 108 to less than 20.

Table 6.2: Results for TPC-DS Q27

	Overheads Metric			MSO Metric		
	Retained Dimensions	MSO	Overheads (Opt calls)	Retained Dimensions	MSO	Overheads (Opt calls)
PSS	9	108	0.5 trillion	9	108	0.5 trillion
Schematic Removal	7	70	528 (<1 sec)	7	70	160128 (36 secs)
MaxSel Removal	1	70		4	37.5	
WeakDim Removal	-	-		2	20	

We shall now see in detail each component of the `DimRed` pipeline respectively in the following three sections.

6.4   Schematic Removal of Dimensions

This component is based on the observation that using standard meta-data structures such as histograms, and physical schema structures such as indexes, it is feasible to establish the selectivities of some of the query predicates with complete or very high accuracy. Further, even if an almost-precise value cannot be established, the metadata could serve to provide tighter lower and upper bounds for selectivities as compared to the default $(0, 1]$ range – these bounds can be leveraged by the subsequent `MaxSelRemoval` stage of the `DimRed` pipeline. We hasten to add that while what we describe here is largely textbook material, we include it for completeness and because of its significant reduction impact on typical OLAP queries, as highlighted in our experimental results of Section 6.7.

For starters, consider the base case of a filter predicate on an ordered domain, a very common occurrence in OLAP queries, whose selectivity analysis can be carried out as follows for equality and range comparisons, respectively.

Equality Predicates: Database engines typically store the *exact* frequency counts for the most com-

monly occurring values in a column. Therefore, if the equality predicate is on a value in this set, the selectivity estimate can be made accurately. On the other hand, values outside of this set will be associated with some bucket of the column's histogram. Therefore, the selectivity range can be directly bounded within $[0, BucketFrequency]$.

An alternate approach to selectivity estimation is to use, if available, an index on the queried column. This is *guaranteed* to provide accurate estimates, albeit at higher computational cost arising out of index traversal. However, since the typical running times for OLAP queries are in several minutes, investing a few seconds on such accesses appears to be an acceptable tradeoff, especially given that choosing wrong plans due to incorrect estimates could result in arbitrary blowups of the response time.

Range Predicates: In this case, histograms can be easily leveraged to obtain tighter bounds, especially with equi-depth histogram implementations. Specifically, the lower bound for the selectivity range is the summation of the frequencies of the buckets that entirely fall within the given range, and the upper bound is the summation of the frequencies of the buckets that partially or completely overlap with the given range.

Similar to equality predicates, if an index is available on the predicate column, then accurate estimates are guaranteed while incurring the index traversal costs.

String predicates: The traditional meta-data structures for string queries often do not yield satisfactory accuracy for selectivity estimation, and typically tend to under-estimate the expected cardinalities. Moreover, they are not particularly useful for obtaining tight lower and upper bounds. Hence, we leverage the strategy described in [LNS07], where summary structures based on *q-grams* are proposed for storing string-related metadata. But for combining the selectivities of the individual sub-string predicates and obtaining deterministic tight bounds, we make use of the *Short Identifying Substring Hypothesis* stated and applied in [CGG04, LNS09]. We also hasten to add that even though these aforementioned strategies provide point estimates for the selectivities, we just adopt their mechanisms for providing bounds on the selectivity range.

The above discussion was for individual predicates. However, in general, there may be multiple filter predicates on a base relation. In such cases, we first compute the ranges or values for each individual predicate (in the manner discussed above), and then use these individual bounds to compute bounds on the relational selectivity as a whole. For instance, when there are conjunctive predicates on a relation, the upper bound on the relational selectivity is simply the upper bound of the least selective predicate. Analogously, for disjunctive predicates, the lower bound is simply the maximum lower bound among the individual predicates. After determining the bounds of the relational selectivity, we

vary the individual predicates between $[0, 1]$, discarding any selectivity combinations that violate the relational selectivity bounds.

6.5 MaxSel Removal of Dimensions

After the schematic removal of dimensions is completed, we know that the optimizer may not be able to accurately estimate the selectivities of the remaining dimensions. However, it may still be feasible to consider some of these selectivities as *"don't-cares"* (i.e. $*$ in regex notation), which is equivalent to removing the dimensions, while provably continuing to maintain overall MSO-safety. The systematic identification and removal of such don't-care dimensions is carried out by the `MaxSelRemoval` module – it does so by the simple expedient of assigning the maximum selectivity (typically, 1) to the candidate dimensions. This maximal assignment *guarantees*, courtesy the PCM assumption, that the time budgets subsequently allocated by `SpillBound` to the bouquet plans in the reduced space are sufficient to cover *all* selectivity values for these dimensions. What is left then is to check whether these deliberately bloated budgets could result in a violation of MSO-safety – if not, the dimensions can be removed.

We describe the operation of the `MaxSelRemoval` module in the remainder of this section. For ease of understanding, we first consider the baseline case of a 2D `PSS`, and then extend the design to higher dimensions. This is followed by an analysis of the module's algorithmic efficiency. Further, for ease of notations we assume D dimensions are retained post the `SchematicRemoval` phase.

6.5.1 Baseline Case: 2D Selectivity Space

Consider a 2D `PSS` with dimensions X and Y, as shown in Figure 6.3, and let the actual (albeit unknown) selectivity of the query in this space be an arbitrary location $q_a(x, y)$. Now assume that we wish to establish whether the X dimension can be dropped in an MSO-safe manner. We begin by projecting q_a to the extreme X-boundaries of the `PSS`, that is, to $X = 0$ and $X = 1$, resulting in $q_a^{min} = (0, q_a.y)$ and $q_a^{max} = (1, q_a.y)$, respectively, as shown in Figure 6.3.

Next, we compute a bound on the sub-optimality of using `SpillBound` with bloated time budgets based on q_a^{max}.

Lemma 6.1 *The sub-optimality for q_a is at most $4 * \frac{\text{Cost}(q_a^{max})}{\text{Cost}(q_a^{min})}$ after removing dimension X from the 2D `PSS`.*

Proof: Let the total execution cost incurred by the `SpillBound` algorithm be denoted by
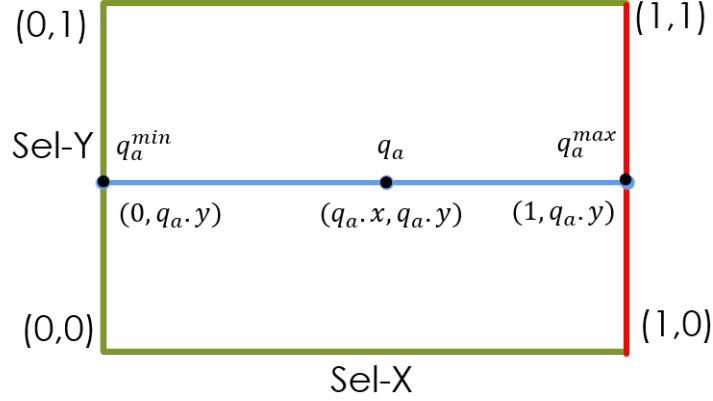
Figure 6.3: Example 2D $\textsc{pss}$

$Cost_{SB}(q_a^{max})$. Then, its sub-optimality is given by

$$SO_{SB} = \frac{Cost_{SB}(q_a^{max})}{\textsc{Cost}(q_a)}$$
$$= \frac{Cost_{SB}(q_a^{max})}{\textsc{Cost}(q_a^{max})} * \frac{\textsc{Cost}(q_a^{max})}{\textsc{Cost}(q_a)}$$

From Equation 6.1, we know that the MSO of SB for a single dimension is 4, and therefore $\dfrac{Cost_{SB}(q_a^{max})}{\textsc{Cost}(q_a^{max})}$ is also upper-bounded by 4. Hence,

$$SO_{SB} \leq 4 * \frac{\textsc{Cost}(q_a^{max})}{\textsc{Cost}(q_a)}$$

Now by the PCM assumption, we know that $\textsc{Cost}(q_a) \geq \textsc{Cost}(q_a^{min})$. Therefore, we have

$$SO_{SB} \leq 4 * \frac{\textsc{Cost}(q_a^{max})}{\textsc{Cost}(q_a^{min})}$$

$\square$

The ratio $\dfrac{\textsc{Cost}(q_a^{max})}{\textsc{Cost}(q_a^{min})}$ captures the *inflation* in sub-optimality for q_a. But note that q_a can be located anywhere in the 2D space, and we therefore need to find the maximum inflation over *all* possible values of y for q_a, and this is denoted by α_X. Formally, the α_X for removing dimension X is defined as:

$$\alpha_X := \max_{q_a.y \in [0,1]} \frac{\textsc{Cost}(q_a{}^{max})}{\textsc{Cost}(q_a{}^{min})}$$

This leads us to the generalization:

77

Corollary 6.1 *After removing a single dimension d from a 2D* `PSS`*, the MSO increases to at most* $4 * \alpha_d$.

Now all that remains is to check whether MSO-safety is retained in spite of the above inflation – specifically, whether

$$4 * \alpha_d \leq 10$$

If the check is true, dimension d can be safely removed from the `PSS` after assigning it the maximum selectivity. (The value 10 comes from the MSO bound for 2D in Equation 6.1.)

Testing Overheads

We now turn our attention to the computational effort incurred in the dimension removal testing procedure. Note that calculating α_d only requires knowledge of the boundaries $sel(d) = 0$ and $sel(d) = 1$, and therefore only $4r$ optimization calls are required in total for testing both dimensions in a 2D space. This is in sharp contrast to the r^2 calls that would have been required for a POSP overlay of the complete `PSS`.

6.5.2 Extension to Higher Dimensions

We now move on to calculating the maximum inflation in sub-optimality for the generic case of removing k dimensions, $s_1, \ldots, s_k$ from a D-dimensional `PSS`, while maintaining MSO-safety. For this, the key idea, analogous to the 2D case, is given a q_a in the `PSS`, find the cost ratios between when the selectivities of the $s_1, \ldots, s_k$ dimensions of q_a are all set to 1, and when they are all set to 0. Since q_a can be located anywhere in the `PSS`, these cost ratios have to be computed for *all* possible selectivity combinations of the retained dimensions. Finally, the maximum of these values gives us $\alpha_{s_1,\ldots,s_k}$. That is,

$$\alpha_{s_1,\ldots,s_k} = \max_{\forall (q.j_{k+1},\ldots,q.j_D) \in [0,1]^{D-k}} \frac{\text{COST}(q^{max})}{\text{COST}(q^{min})} \tag{6.2}$$

with q^{max} and q^{min} corresponding to the locations with $q.i = 1$ and $q.i = 0$, respectively, $\forall i \in \{1,..,k\}$.

To illustrate the above, consider the example 3D `PSS` with dimensions X, Y and Z shown in Figure 6.4 (a). Here, if we wish to consider dimension Y for removal, we need to first compute the sub-optimality inflations across all matching pairs of points in the (red-colored) 2D surfaces corresponding to $Y = 0$ and $Y = 1$. Then, α_Y is given by the maximum of these inflation values.

Finally, to determine whether the removal of the $s_1, \ldots, s_k$ dimensions is MSO-safe, the check is simply the following:

$$\alpha_{s_1,\ldots,s_k} * MSO_{SB}(D-k) < MSO_{SB}(D) \tag{6.3}$$

78

(a) α_Y from Surface   (b) α_Y from Corners
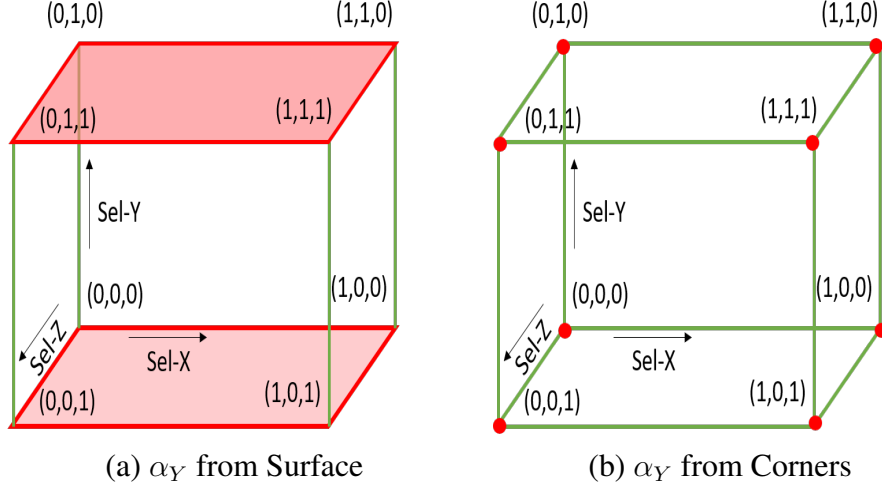
Figure 6.4: 3D `PSS` - Calculation of α_Y

Testing Overheads:

The computational efforts incurred, in terms of optimizer calls, for calculating $\alpha_{s_1,\dots,s_k}$ is $\theta(2^D * r^{D-1})$. This is because the sub-optimality inflations need to be calculated for *every* selectivity combination of the retained dimensions, making the testing overheads to grow exponentially with the `PSS` dimensionality. It may therefore appear, at first glance, that we have merely shifted the computational overheads from the exhaustive POSP overlay to the modules of the `DimRed` pipeline. However, as is shown next, it is feasible, with mild assumptions, to design an efficient mechanism to compute α_M.

6.5.3 Efficient Computation of `MaxSelRemoval`

To provide efficiency in the `MaxSelRemoval` algorithm, we bring in two techniques, *Greedy Removal* and *Corner Inflation*. With Greedy Removal, we do not exhaustively consider all possible groups of dimensions for removal. Instead, we first compute for each dimension d, its individual α_d assuming that just d is removed from the `PSS` and all other dimensions are retained. Based on these inflation values, a sequence of dimensions is created in increasing α_d order. Then, we iteratively consider *prefixes* of increasing length from this sequence with the stopping criterion based on the objective – overheads minimization or MSO minimization.

As an example, we show in Figure 6.5 the resultant MSOs for the greedy removal of dimensions for TPC-DS Q7, which has an 8-dimensional `PSS`. As can be seen in the figure, the MSO initially declines from its starting value of 88 as we keep removing the low α_d dimensions. However, after a certain point, it begins to rise again. When overheads minimization is the objective, the algorithm will remove the dimension group {*d4, d5, d6, d7, d0, d1, d2*}, and then stop due to violation of MSO-

safety. By this time, the ESS dimensionality is reduced to just 1, and the MSO guarantee is 70. On the other hand, when MSO minimization is the objective, the minimum MSO of 33 is obtained by removing dimensions {*d4, d5, d6, d7*} with a combined inflation factor of just **1.17**!
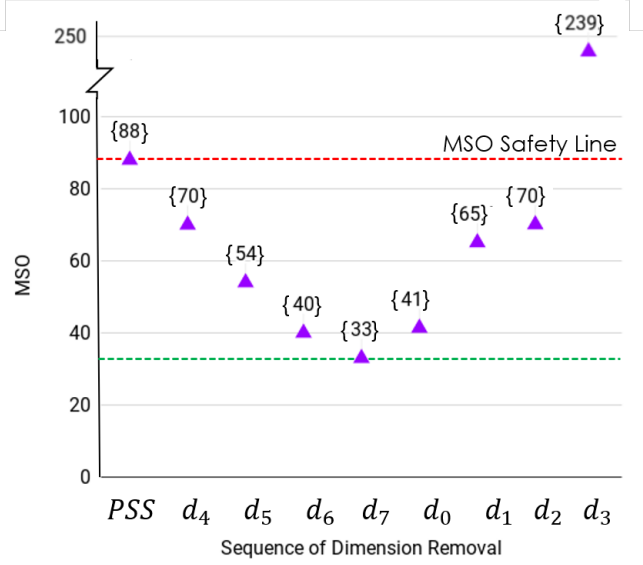
Figure 6.5: MSO Profile for Greedy Dimension Removal

Although the greedy removal strategy does significantly lower the overheads, it still requires $\mathcal{O}(D * r^{D-1})$ optimizer calls since the α_d has to be calculated for every dimension d. This is where we bring in our second strategy of *Corner Inflation*. Specifically, we assume that the α_d (for every dimension d) is always located at one of the *corners* of the PSS and not in the interior – if this is true, then the number of optimizer calls required to calculate for all α_ds is reduced to $\theta(2^D)$, which is *independent* of the resolution. We have empirically verified, as highlighted in Section 6.7, that this assumption is is generally valid. Moreover, in the remainder of this section, we formally prove that, under some mild assumptions, the corner location of the α_d is only to be expected.

6.5.4 Proof of Corner Inflation

For ease of exposition, we first analyze a 3D PSS, and later generalize the proof to higher dimensions. Consider the 3D PSS shown in Figure 6.4 (b), with dimensions X, Y and Z, and dimension Y being the candidate for removal. Our objective is to show that optimization calls are required only along the corners in the figure, and not along the surface walls (unlike Figure 6.4 (a)). To start with, we introduce an inflation function, f, that captures the sub-optimality inflation as a function of the selectivity combinations of the retained dimensions X and Z, along the extreme values of the removed dimension

Y. Formally, the inflation function, $f(x, z)$, is defined as follows:

$$f(x, z) = \frac{\text{COST}(q.x, q.y = 1, q.z)}{\text{COST}(q.x, q.y = 0, q.z)}$$

Behavior of function f

To analyze f's behavior, we leverage the notion of optimal cost surface (OCS), which captures the cost of the optimal plan at every location in the PSS. For now, assume that the OCS exhibits *axis-parallel linearity* (APL) in X, Y and Z. That is, the OCS is of the form:

$$OCS(x, y, z) \quad = \quad u_1 x \; + \; u_2 y \; + \; u_3 z \; + \; u_4 xy \; + \; u_5 yz \; + \; u_6 xz \; + \; u_7 xyz \; + \; u_8 \quad (6.4)$$

where the u_i are arbitrary scalar coefficients.

When dimension $Y = 1$, the projected OCS with this y value is represented as

$$OCS|_{y=1} = a_1 x + a_2 z + a_3 xz + a_4, \tag{6.5}$$

where the a_i are the new scalar coefficients. Analogously, when $Y = 0$, the projected OCS becomes

$$OCS|_{y=0} = b_1 x + b_2 z + b_3 xz + b_4 \tag{6.6}$$

Thus f can now be rewritten as a point-wise division of a pair of 2D APL functions:

$$f(x, z) = \frac{OCS|_{y=1}}{OCS|_{y=0}} = \frac{a_1 x + a_2 z + a_3 xz + a_4}{b_1 x + b_2 z + b_3 xz + b_4} \tag{6.7}$$

Now consider the function $f_z(x)$, which keeps dimension Z constant at some value and varies only along dimension X. When $Z = z_0$, $f_{z_o}(x)$ which is essentially division of two lines, using which we show the following lemma.

Lemma 6.2 *Given a line segment $Z = z_o$ that is parallel to the X axis, the maximum sub-optimality occurs at one of the end points (($0, z_o$) and ($1, z_o$)).*

Proof: In order to prove this, we use a crucial observation that the function value of $f'_{z_o}(x)$ monotonically increases or decreases. Thus, showing that the maximum sub-optimality occurs at the end points. Consider,

$$f_{z_o}(x) = \frac{cx + d}{px + q} \tag{6.8}$$

81

where $c = a_1 + a_3 z_0$, $d = a_2 z_0 + a_4$, $p = b_1 + b_3 z_0$, and $q = b_2 z_0 + b_4$. Taking the derivative of the above function wrt x, we get

$$f'_{z_o}(x) = \frac{c(px + q) - p(cx + d)}{(px + q)^2} = \frac{cq - dp}{(px + q)^2} \tag{6.9}$$

The above equation shows that the sign of derivative is constant wrt varying x. This means that the function $f_{z_o}(x)$ is either decreasing or increasing. Clearly the value of the inflation function is maximum at either of the end-points. □

Putting together all this machinery leads to the following lemma:

Lemma 6.3 *Computing α_Y along the corners of an* PSS *is sufficient to establish α_Y within the entire* PSS.

Proof: From the previous lemma, we know that for the lines segments $Z = z, z \in (0, 1)$ that are parallel to X-axis, the local α_Y occurs at one of the end-points of these line segments. By induction, the local α_Y computation can be moved to the corners of the PSS. □

Relaxing the APL assumption

In the above analysis, we assumed that the OCS follows the *APL* property (Equation 6.4), but this may not always hold in practice. Thus, we now extend the previous result by relaxing the linearity assumption to its piece-wise equivalent [HS02], and assuming that these linear pieces follow a certain empirically validated slope behaviour. The idea now is to divide the domain of the OCS into its constituent APL-compliant pieces. Later, we show that these α_Y's can be moved across pieces to finally end up at the corners of the original PSS. This leads us to the following main theorem:

Theorem 6.1 *Computing α_Y along the corners is sufficient to establish α_Y within the entire* PSS.

Proof: We saw in Lemma 6.3 that if OCS follows APL assumption, then computation at the corners of the PSS is sufficient to calculate α_Y. With OCS following piecewise APL property (presented next), let us say that there are at most k pieces in any 1D segment of the PSS.

Then, for each of the $2k$ pieces conditioned on the two segments $Z = z_0$ at $OCS|_{y=1}$ and $OCS|_{y=0}$, we make a mild assumption that for these $2k$ pieces the slope of the inflation function is monotonically increasing or deceasing. This assumption is assessed exhaustively over thousands of such 1D segments (chosen from our suite of queries), by evaluating if $cq \geq dp$ or $cq \leq dp$ (numerator in equation 6.9) is consistently true across all the above $2k$ pieces. The assessment shows that it is indeed true in more than 90% of the 1D segments in the ESS, on an average across queries, along all such $2k$ pieces. Thus we can claim that the α_Y would still lie at one of the corners of an ESS for practical settings. □

Piecewise APL Fit of OCS Let us now discuss the accuracy of OCS to a Piecewise APL function. It is important point to note here is that we do not require an accurate *quantitative fit*, but only an accurate *qualitative fit* – that is, the slope behavior should be adequately captured.

(a) Original OCS   (b) Partitioned OCS Domain   (c) OCS fitted with piecewise functions
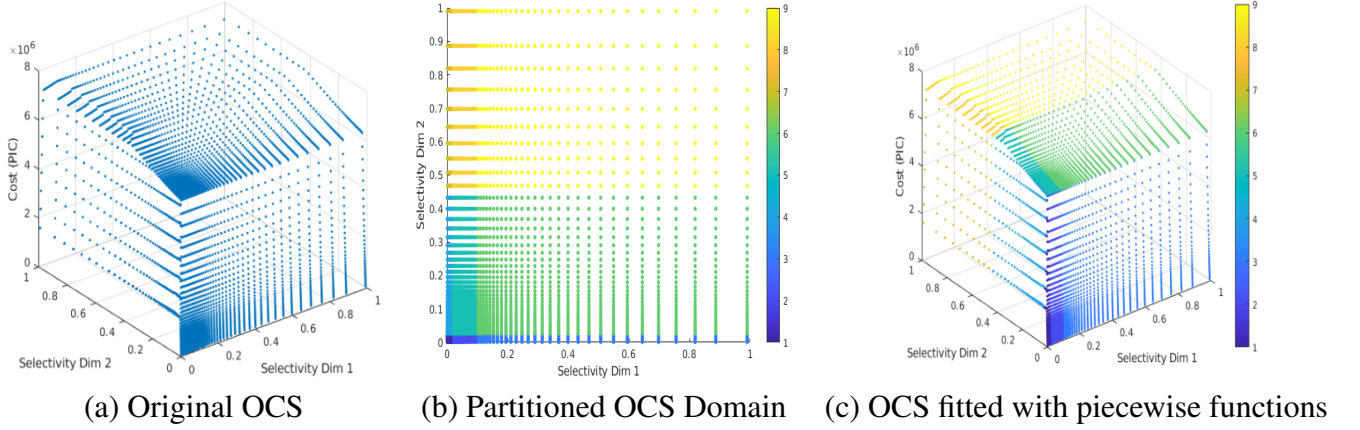
Figure 6.6: Original OCS and OCS fitted with an *APL* function per region of the partitioned input domain

Let us consider an example OCS of the TPC-DS query 26, generated using repeated invocations of the *PostgreSQL* optimizer. This is shown in Figure 6.6 (a). The 2D input domain of the OCS, which is the 2D selectivity region spanned by dimensions 1 and 2 is divided into 9 regions. Each region is then fitted with the 2D *APL* function of the form,

$$f(x, y) = ax + by + cxy + d \tag{6.10}$$

We use non-linear least squares regression to fit the function and we are able to do so with normalized RMSE = 9%. The projection of the boundaries of these regions on the input domain is shown in Figure 6.6 (b).

The problem now is identification of these regions where the Function 6.10 fits nicely. This is done using the K-subspace clustering methods for 2D and higher dimensional planes as described in [WDL09] shown in Figure 6.6 (c).

Extension to Higher Dimensions

Our above analysis of the Corner Inflation procedure was carried out for a 3D PSS. For handling a higher dimension PSS, the process follows by induction wherein for every 1D segment α_d is moved to its end points. Hence, by induction we can show that α_d occurs at the PSS corners.

83

6.6 WeakDimRemoval techniques

The dimensions retained post `SchematicRemoval` and `MaxSelRemoval` techniques are the dimensions for which the `ESS` is constructed, that we assume to be D in number for the ease of notations. After the `ESS` construction the *isocost contours* are identified for `SpillBound`'s (SB) execution. Here, an isocost contour, corresponding to cost C, represents the connected selectivity curve along which the cost of the optimal plan is C. A sample contour can be seen in Figure 6.7 (a) shown as colored 1D curve. Series of contours starting from the lowest to highest cost in `ESS`, with geometric progression of two, are constructed. The key idea in SB is to perform D plan executions per contour from the lowest cost contour, until the actual selectivities of all the `epps` are explicitly learnt. Finally, the optimal plan is identified and executed for query completion. In this section we show how can we reduce the number of plan executions per-contour from D, to attain a tighter MSO. Again for ease of exposition, we consider the base case of $D = 2$, with `epp` X and Y. Then, move on to the 3D scenario to present the subtleties of the algorithm, from where it can easily be generalized to arbitrary dimensions.

6.6.1 `WeakDimRemoval` 2D scenario

We use the sample 2D `ESS`, shown in Figure 6.7 (a), for ease of exposition of the algorithm. The figure depicts the isocost contour $\mathcal{IC}_i$, associated with cost $\texttt{CC}_i$, and annotated with the optimal plans P_1, P_2, P_3 and P_4. Note that the cost of these four plan on the contour locations costs $\texttt{CC}_i$. In SB each of these contour plans tries to individually and incrementally learn selectivities of the two `epps`. SB carefully assign an `epp` for a contour plan to learn its selectivity, in order to achieve MSO guarantees. This mode of execution of plans trying to learn individual `epp` selectivities is referred to as *spillmode execution of the plan while spilling on the* `epp`. For instance, in Figure 6.7 (a), the plan P_1 is annotated as P_1^y to indicate that it spills on the `epp` Y during execution and learn its selectivity.

Contour Plan's Learning `epp` in SB

For each plan on the contour $\mathcal{IC}_i$, SB chooses an `epp` on which that plan needs to be *spilled*, so that the cost-budget for the plan is utilized to maximally learn the selectivity of that predicate only. This choice is based on the plan structure and the ordering of its constituent pipelines. We use this critical component of SB for our `WeakDimRemoval` technique, which is described next.

The 2D Algorithm

As mentioned before, SB requires at most two executions per contour until all the actual selectivity for both the `epps` are learnt. Let us say that dimension X is removed (we discuss this choice later) using the 2D `WeakDimRemoval` algorithm. The idea in the algorithm, is to *piggyback* X's plan execution

(a) Contour on a 2D `ESS`   (b) P_1 replacing other plans
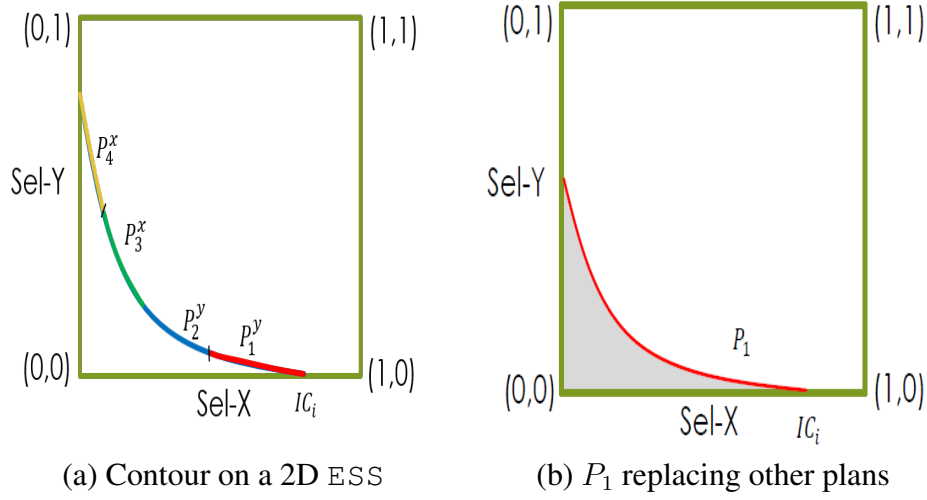
Figure 6.7: `WeakDimRemoval` for $D = 2$

(and, hence its selectivity learning) along with Y-spilling plans. This is achieved by considering all plans that are X-spilling to be Y-spilling, in short, by ignoring the error-prone X-predicate in the pipeline order. Thus, we end up in all plans in a contour to be Y spilling. Finally, in order to make `WeakDimRemoval` MSO-efficient, we choose a plan on the contour which is relatively the cheapest on all locations of the contour, which is captured by an inflation factor. The maximum of these inflation factor's across all the contours is captured by β_X.

Now, if β_X is low such that $MSO_{SB}(1) * (1 + \beta_X) < MSO_{SB}(2)$, then `WeakDimRemoval` is successful for reduction in the MSO. In essence, we say that the *weak* dimension, X's execution is piggybacked by its *strong* dimension counterpart Y.

Algorithm Trace   Let us now trace the above algorithm for our example `ESS` and contour $\mathcal{IC}_i$. First, we choose all the contour plans, which is P_1 to P_4. All these plans are assigned to spill on `epp` Y. Then, each of these four plans, are costed at all the contour locations, in order to find the best one-plan replacement with the least inflation factor. In this scenario, P_1 happens to be our best replacement plan, as shown in Figure 6.7 (b).

Proof of Correctness

The algorithm's correctness, in order to achieve the desired MSO, is primarily dependent on the lower bound of selectivity learning of a plan while piggybacking the executions.

Lemma 6.4 (Piggybacked Execution) *Consider the contour plan P_r which replaces all the plans on contour $\mathcal{IC}_i$ with an cost inflation factor of β_X. Further, let P_r is assigned to spill on Y and executed with budget $CC_i(1 + \beta_X)$. Then, then we either learn: (a) the exact selectivity of Y, or (b) infer that*

q_a lies beyond the contour.

Proof: $\mathcal{IC}_i$ represents the set of points in the ESS having their optimal cost equal to $\mathtt{CC_i}$. The cost of all points $q \in \mathcal{IC}_i$ is at most $\mathtt{CC_i}(1 + \beta_X^i)$ when costed using P_r. Now when the plan P_r is executed in the spill-mode with cost budget $\mathtt{CC_i}(1 + \beta_X^i)$ it may or may not complete.

For an internal node N of a plan tree, we use $N.cost$ to refer to the execution cost of the node. Let N_Y denote the internal node corresponding to Y in plan P_r. Partition the internal nodes of P_r into the following: $Upstream(N_Y)$, $\{N_Y\}$, and $Residual(N_Y)$, where $Upstream(N_Y)$ denotes the set of internal nodes of P_r that appear before node N_Y in the execution order, while Residual (N_Y) contains all the nodes in the plan tree excluding $Upstream(N_Y)$ and $\{N_Y\}$.Therefore,

$$Cost(P_r, q) = \sum_{N \in Upstream(N_Y)} N.cost + N_Y.cost + \sum_{N \in Residual(N_Y)} N.cost$$

Case-1 : The value of the first term in the summation $Upstream(N_Y)$ is known with certainty if it does not contain N_X. Further, the quantity $N_Y.cost$ is computed assuming that the selectivity of N_Y is $q.y$ for any point $q \in \mathcal{IC}_i$ with maximum sub-optimality of β_X^i . Since the output of N_Y is discarded and not passed to downstream nodes, the nodes in $Residual(N_Y)$ incur zero cost. Thus, when P_r is executed in spill-mode, the budget $\mathtt{CC_i}(1+\beta_X^i)$ is sufficiently large to either learn the exact selectivity of Y (if the spill-mode execution goes to completion) or to conclude that $q_a.y$ is greater than $q.y$, $\forall q \in \mathcal{IC}_i$, since P_r is costed for all $q \in \mathcal{IC}_i$. Hence, q_a lies beyond the contour $\mathcal{IC}_i$.

Case-2 : Now if N_X is contained in $Upstream(N_Y)$ then its cost is not known with certainty, however since P_r is costed for all $q \in \mathcal{IC}_i$, all the selectivity combinations of $(q.x, q.y)$, $\forall q \in \mathcal{IC}_i$ get considered. Hence, for all these combinations the sum of the quantity $\sum_{N \in Upstream(N_Y)} N.cost + N_Y.cost \leq$ $\mathtt{CC_i}(1 + \beta_X^i)$. Similar to Case-1, the output of N_Y is discarded and not passed to downstream nodes, hence the nodes in $Residual(N_Y)$ incur zero cost. Thus, when P_r is executed in spill-mode, the budget is sufficiently large to either learn the exact selectivity of Y and X (if the spill-mode execution goes to completion) or to conclude that $q_a \succ q$ (strictly dominates) for any $q \in \mathcal{IC}_i$ which implies that $Cost(P_{q_a}, q_a) > \mathtt{CC_i}$ i.e it lies beyond the contour by *PCM*. $\qquad\square$

Let there be $m = \log_2\left(\frac{C_{max}}{C_{min}}\right)$ number of contours, let P_i be the best 1-plan replacement with sub-optimality β_X^i for each contour $\mathcal{IC}_i$ from $i = 1 \rightarrow m$. Let $\beta_X = \max_{i=1\rightarrow m} \beta_X^i$.

Lemma 6.5 *The MSO for the 2D scenario when contour plan replacement is done along a single dimension X is $4(1 + \beta_X)$.*

Proof: The query processing algorithm executes the best 1-plan replacement, P_i, for each contour $\mathcal{IC}_i$, starting from the least cost contour. Each execution of P_i is performed with an inflated budget of $\texttt{CC}_\texttt{i}(1 + \beta_X)$. Since each contour now has only 1 plan with fixed inflated budget, using the 1D SB algorithm with inflated contour budgets it is easy to show that the MSO for the 2D scenario post $\texttt{WeakDimRemoval}$ is equal to $MSO_{SB}(1) * (1 + \beta_X) = 4 * (1 + \beta_X)$. □

It is important to note that β_X - which denotes the worst case sub-optimality incurred for making plan replacements along the dimension X is a function of the dimension X itself.

Hence, By doing piggybacked executions of *'weak'* dimensions (dimensions with low α) along the *'strong'* dimensions (dimensions with high α), $\texttt{WeakDimRemoval}$ makes the MSO a function of impactful dimensions only.

6.6.2 $\texttt{WeakDimRemoval}$ 3D Scenario

In this sub-section we see how the $\texttt{WeakDimRemoval}$ technique can be extended to the 3D scenario, consisting of dimensions X, Y and Z, where we wish to do $\texttt{WeakDimRemoval}$ along dimension X. As in the 2D scenario, all the plans on the contour become either Y-spilling or Z-spilling by ignoring the epp X in the pipeline order. Let the set of plans which were originally X-spilling plans, but now considered as either Y-spilling or Z-spilling, be denoted by P^T.

The main idea of the algorithm as stated earlier is to execute two plans (one for each strong dimension) and piggyback the execution of the weak dimension along with these strong ones. In our case, the execution of X is piggy backed with Y and Z. Let q^x_{sup}, q^x_{inf} denote the points having maximum and minimum X-selectivity on the contour respectively. Also, let $\mathsf{Sup}_x = q^x_{sup}.x$ and $\mathsf{Inf}_x = q^x_{inf}.x$. Let us first characterize the geometry of the contours based on the minimum and maximum selectivities of the replaced dimension X, captured by $X = \mathsf{Inf}_x$ and $X = \mathsf{Sup}_x$ respectively. There are three possibilities:

1. a 2D contour line on the $X = \mathsf{Inf}_x$ slice and a point on $X = \mathsf{Sup}_x$ slice

2. a 2D contour line on the $X = \mathsf{Sup}_x$ slice and a point on $X = \mathsf{Inf}_x$ slice

3. a 2D contour line on both $X = \mathsf{Inf}_x$ and $X = \mathsf{Sup}_x$ slices

The rest of the section and figures correspond to the Case 1, but all the Lemmas and Theorems are easily generalizable for all the cases mentioned above.

To piggyback X's execution with Y, consider a point q' on the $X = \mathsf{Inf}_x$ slice, let its coordinates be such that $q' = (\mathsf{Inf}_x, y', z')$. This is shown in Figure 6.8. Let us define the set $S_{y'} := \{q | q \in \mathcal{IC}_i$ and $q.y \leq y'\}$, that contains all the (x, y) selectivity combinations pertaining to the contour such

that $y \leq y'$. We now construct the minimal (x, y)-dominating set, that spatially dominates all points in $S_{y'}$ denoted by $\hat{S}_{y'}$. Formally,

$$\hat{S}_{y'} := \forall q \in S_{y'}, \exists \hat{q} \in \hat{S}_{y'} \text{ such that } (\hat{q}.x, \hat{q}.y) \succeq (q.x, q.y) \tag{6.11}$$

(a) Choice of a Plan for a point q' (b) Choice of P^z Plan for a point q' (c) Choice of P^y Plan for a point q'
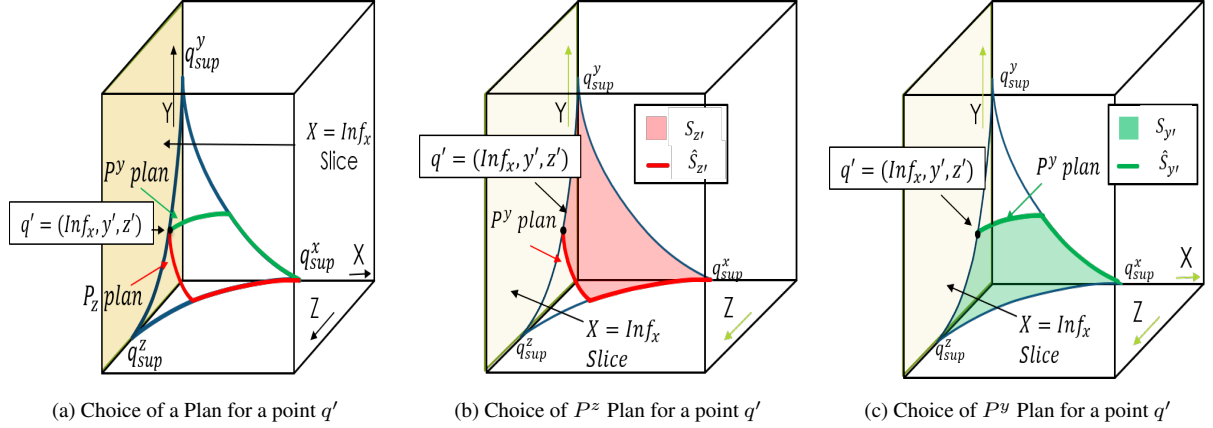
Figure 6.8: `WeakDimRemoval` 3D Scenario Phase 1

6.6.3 `WeakDimRemoval` **Overheads**

Let the maximum number of plans on any contour be denoted by ρ, the maximum number of contours be m. The `ESS` dimensionality post `SchematicRemoval` and `MaxSelRemoval` is $D - k_r$, which makes the size of the contour r^{D-k_r-1}. Then the effort required to do `WeakDimRemoval` is of the order $O(\rho * m * r^{D-k_r-1})$ Abstract Plan Costing (APC) calls. The APC calls are typically at least 100 times faster than usual optimizer calls, since the optimizer does not need to come up with a plan for the given location, instead it just needs to cost the *specified plan* using its cost model which makes it extremely economical. By doing an intra-contour anorexic plan reduction [DDH07], we have $\rho_{anorexic}$: the maximum number of reduced plans on any contour, which is typically less than 10.

$$
\begin{aligned}
\frac{Overheads((\text{PR}))}{Overheads(SB)} &= \frac{(\rho_{anorexic} * m * r^{D-k_r-1}) * \text{APC calls}}{r^{D-k_r} * \text{OPT calls}} \\
&= \frac{(\rho_{anorexic} * m * r^{D-k_r-1}) * 10^{-2}}{m * r^{D-k_r-1}} \\
&= \frac{\rho_{anorexic} * m}{r * 100} \\
&\leq \frac{1}{r} \text{(For typical values of } \rho_{anorexic} \text{ and } m)
\end{aligned}
$$

This makes the overheads of `WeakDimRemoval` just 1% of the overall required compile time effort for the resolution $r = 100$, and 5% for $r = 20$.

6.7 Experimental Evaluation

Having described the `DimRed` technique, we now turn our attention to its empirical evaluation. Unlike `SpillBound` class of techniques which provide platform-independent performance guarantees, `DimRed` does not provide guarantees on the number of dimensions that can be removed for a query. Hence, here we increase the scope of performance evaluation to 37 TPC-DS and 21 JOB queries. Moreover, the JOB queries feature complex filter predicates involving string comparisons with the `LIKE` operator, and multiple predicates on a single base relation.

6.7.1 Goodness of OCS Surface Fit

To measure the goodness of the fit we compute the *Normalized RMSE* and *Normalized Max Error*. The results in the Tables 6.3 and 6.4 show that we are able to achieve good fitting and hence validate our claims.

6.7.2 Validation of Corner Inflation

As discussed in Section 6.5, an efficient implementation of `MaxSelRemoval` requires the α_d corresponding to any dimension d to be located on the *corners* of the `PSS` itself. We have conducted a detailed validation of this behavioral assumption. Specifically, we carried out an offline exhaustive construction of the complete `PSS` and calculated the α_d for each dimension d. These values were then compared with the αs obtained by restricting the calculation to only the corners' of the `PSS`. The results showed that for most dimensions in the queries in our workload, α did occur on the corners of the `PSS`.

6.7.3 Overheads Minimization Objective

We now turn our attention to the `DimRed` performance on the overheads minimization metric, where the objective is to minimize the `PSS` dimensionality while retaining MSO-safety. The performance results for this scenario are shown in Figures 6.9 (a) and 6.9 (b) for the TPC-DS and JOB query suites, respectively. In these figures, the full height of each vertical bar shows the original `PSS` dimensionality, while the bottom segment (blue fill) within the bar indicates the final `ESS` dimensionality, after reduction by the `SchematicRemoval` (yellow checks) and `MaxSelRemoval` (green braid) modules.

The important observation here is that across all the queries, containing as high as 19 dimensions

Table 6.3: RMSE (TPC-DS)

Query Number	Surface Fit	
	Normalized RMSE	*Normalized Max Error*
Q03	11.60	24.51
Q07	15.79	22.16
Q12	16.50	16.56
Q15	10.62	16.63
Q18	10.48	21.79
Q19	16.45	22.68
Q21	16.34	27.32
Q22	11.33	14.84
Q26	11.42	22.02
Q27	16.94	27.32
Q29	15.20	16.33
Q36	8.47	23.15
Q37	9.58	23.29
Q40	15.48	21.48
Q42	9.09	26.96
Q43	14.99	22.42
Q52	15.14	21.70
Q53	14.39	21.32
Q55	9.03	14.72
Q62	15.78	22.78
Q63	9.90	10.28
Q67	10.19	11.21
Q73	14.52	19.30
Q82	10.56	17.36
Q84	16.78	27.29
Q86	15.35	19.54
Q89	10.59	19.92
Q91	16.42	22.22
Q96	14.17	27.84
Q98	14.20	15.70
Q99	14.88	15.63

Table 6.4: RMSE (JOB)

Query Number	Surface Fit	
	Normalized RMSE	*Normalized Max Error*
Q01	15.30	20.62
Q08	11.86	15.82
Q09	12.40	13.65
Q10	14.31	22.79
Q11	12.27	19.12
Q12	11.18	16.13
Q13	7.91	10.14
Q14	13.06	13.54
Q15	15.43	16.74
Q16	16.77	25.68
Q19	15.11	24.27
Q20	10.19	16.54
Q21	9.84	19.68
Q22	8.77	19.29
Q23	12.91	17.63
Q24	11.36	17.39
Q25	11.72	11.83
Q26	9.23	14.11
Q28	12.46	17.19
Q29	9.11	11.88
Q33	11.83	20.65

(a) TPC-DS Queries

(b) JOB Queries
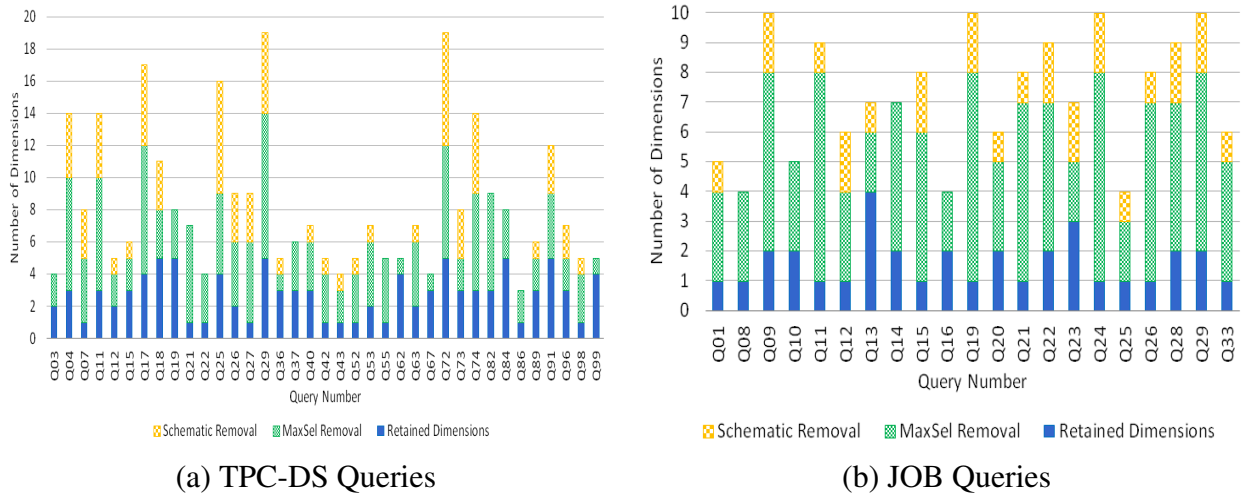
Figure 6.9: Dimensionality Reduction for Overheads Minimization

of the initial `PSS`, the eventual `ESS` dimensionality is essentially *"anorexic"*, being always brought down to five or less. In fact, for as many as 10 queries in TPC-DS and 11 queries in JOB the number of dimensions retained is just **1**! We also see that `MaxSelRemoval` usually plays the primary role, and `SchematicRemoval` the secondary role, in realizing these anorexic dimensionalities. Inspection of the retained dimensions showed that all the base filter predicates are removed from the `PSS` either by `SchematicRemoval` or by `MaxSelRemoval`, leaving behind only the high-impact join dimensions. Another observation is that `SchematicRemoval` removal is not as successful on the JOB benchmark as on TPC-DS – this is due to the complex filter predicates on the base relations. But by using the bounds provided by `SchematicRemoval`, `MaxSelRemoval` is successfully able to remove all of them with only a small MSO inflation. These results also justify our creation of an automated pipeline to replace the handpicking of dropped dimensions in the earlier literature.

After the above dimensionality reduction, the next step in the `DimRed` pipeline is to try and improve the MSO through invocation of the `WeakDimRemoval` module. The resulting MSO values are shown in Figures 6.10 (a) and 6.10 (b) for the TPC-DS and JOB query suites, respectively. In these figures, the full height of each vertical bar shows the MSO of the original `PSS`, while the bottom segment (blue fill) within the bar indicates the final MSO, after initial improvements by `SchematicRemoval` and `MaxSelRemoval` (yellow-green checks) and subsequently by `WeakDimRemoval` (red lines). The important observation here is that for a majority of the queries, the final MSO is substantially lower than the starting value. For instance, with TPC-DS Q91, the MSO is tightened from 180 to 40, and with JOB Q19, the improvement is from 130 to 35. Overall, we find an average decrease of 54% and 67% for TPC-DS and JOB, respectively.

(a) TPC-DS Queries   (b) JOB Queries
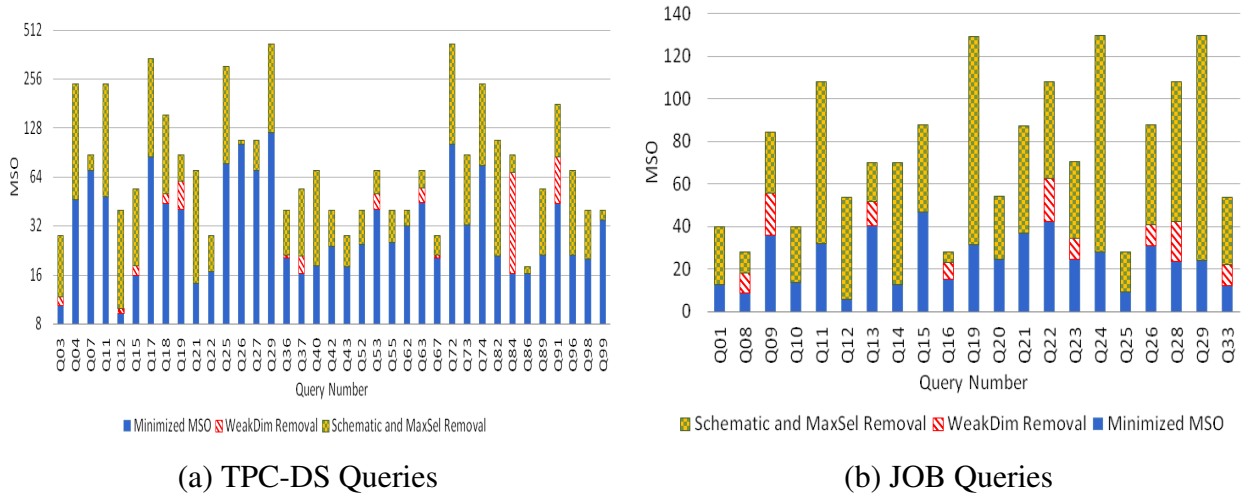
Figure 6.10: MSO Profile for Overheads Minimization

6.7.4   MSO Minimization Objective

We now turn our attention to the goal of dimensionality reduction with the objective of minimizing MSO, subject to the safety requirement. The dimensionality results for this alternative scenario are shown in Figures 6.11 (a) and 6.11 (b) for the TPC-DS and JOB query suites, respectively. In these figures, the full height of each vertical bar shows the original `PSS` dimensionality, while the bottom blue segment within the bar indicates the final `ESS` dimensionality, after reduction by the `SchematicRemoval` (yellow checks) and `MaxSelRemoval` (green braids) modules.

As should be expected, the number of dimensions retained are slightly higher with MSO minimization as compared to overheads minimization. However, all queries still have less than or equal to five dimensions.

The corresponding MSO profile is shown in Figures 6.12 (a) and 6.12 (b) for the TPC-DS and JOB query suites, respectively confirm this claim. Again, the full vertical height captures the original MSO, and the bottom segment (blue fill) shows the final MSO, after improvements due to `SchematicRemoval` (yellow checks), `MaxSelRemoval` (green braid) and `WeakDimRemoval` (red lines).

In summary, for the TPC-DS queries, we obtain an average improvement of 63%, whereas for the JOB queries it is 77%.

6.7.5   Time Efficiency of `DimRed`

A plausible concern about `DimRed` is whether the overheads saved due to dimensionality reduction may be negated by the computational overheads of the pipeline itself. To address this issue, we

(a) TPC-DS Queries   (b) JOB Queries
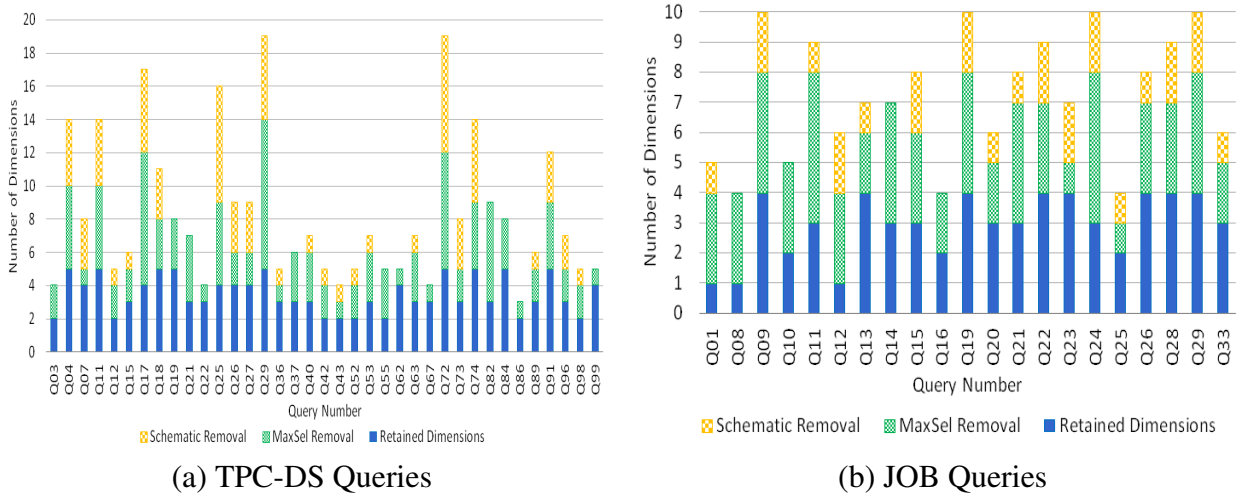
Figure 6.11: Dimensionality Reduction for MSO Minimization

present in Table 6.5, a sample profile of `DimRed`'s efficiency, corresponding to TPC-DS Query 91, which is the highest dimensionality query in our workload, featuring 6 filter and 6 join predicates. In the table, the optimizer calls made by the pipeline, and the overall time expended in this process, are enumerated. We find that the entire pipeline completes in less than 12 minutes, *inclusive* of the POSP overlay on the `ESS`, whereas the compilation efforts on the original `PSS` would have taken more than a year!

Table 6.5: `DimRed` EFFICIENCY: TPC-DS QUERY 91

	Dimensionality	MSO	Overheads (Opt Calls)	Time (Secs)
PSS	12	180	4 quadrillion(10^{15})	> 1 year
Schematic Removal	9	108	2^9(MaxSel) $+ 32 * 10^5$(ESS) $+ 1.6 * 10^5$(WeakDim) $= 33.6 * 10^5$	0.01 (MaxSel) $+ 640$ (ESS) $+ 32$ (WeakDim) ≈ 11 minutes
MaxSel Removal	5	84		
WeakDim Removal	2	44		

Table 6.6: `DimRed` TIME EFFICIENCY: TPC-DS (OVERHEADS MINIMIZATION)

Query Number	Overheads (Optimizer Calls)				Overheads(Time in Secs)			
	MaxSel Calls	*ESS Calls*	*WeakDim Calls*	*Total*	*MaxSel*	*ESS*	*WeakDim*	*Total*
Q03	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q07	32	20	1	53	0.0064	0.0040	0.0002	**0.0106**
Q12	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q15	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q18	256	3200000	160000	3360256	0.0512	640.0000	32.0000	**672.0512**
Q19	256	3200000	160000	3360256	0.0512	640.0000	32.0000	**672.0512**
Q21	128	20	1	149	0.0256	0.0040	0.0002	**0.0298**
Q22	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q26	64	400	20	484	0.0128	0.0800	0.0040	**0.0968**
Q27	512	20	1	533	0.1024	0.0040	0.0002	**0.1066**
Q29	512	3200000	160000	3360512	0.1024	640.0000	32.0000	**672.1024**
Q36	16	8000	400	8416	0.0032	1.6000	0.0800	**1.6832**
Q37	64	8000	400	8464	0.0128	1.6000	0.0800	**1.6928**
Q40	64	8000	400	8464	0.0128	1.6000	0.0800	**1.6928**
Q42	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q43	8	20	1	29	0.0016	0.0040	0.0002	**0.0058**
Q52	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q53	64	400	20	484	0.0128	0.0800	0.0040	**0.0968**
Q55	32	20	1	53	0.0064	0.0040	0.0002	**0.0106**
Q62	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
Q63	64	400	20	484	0.0128	0.0800	0.0040	**0.0968**
Q67	16	8000	400	8416	0.0032	1.6000	0.0800	**1.6832**
Q73	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q82	512	8000	400	8912	0.1024	1.6000	0.0800	**1.7824**
Q84	256	3200000	160000	3360256	0.0512	640.0000	32.0000	**672.0512**
Q86	8	20	1	29	0.0016	0.0040	0.0002	**0.0058**
Q89	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q91	4096	3200000	160000	3364096	0.8192	640.0000	32.0000	**672.8192**
Q96	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q98	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q99	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
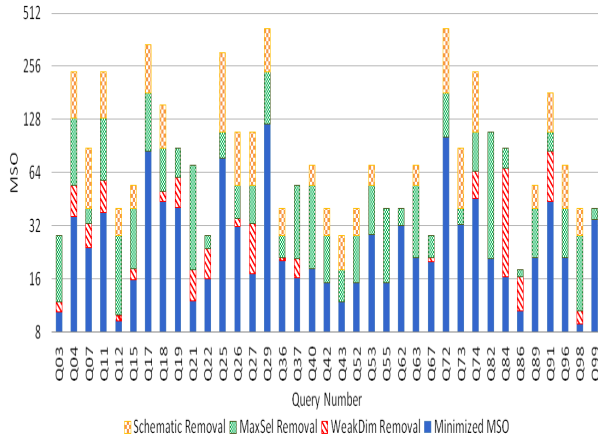
(a) TPC-DS Queries
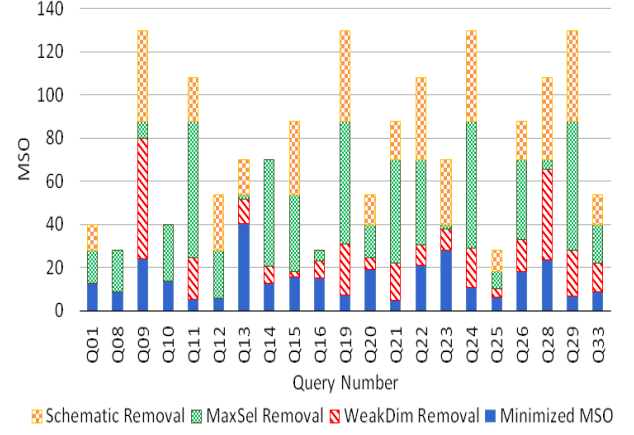
(b) JOB Queries

Figure 6.12: MSO Profile for MSO Minimization

Table 6.7: `DimRed` TIME EFFICIENCY: JOB (OVERHEADS MINIMIZATION)

Query Number	Overheads (Optimizer Calls)				Overheads(Time in Secs)			
	MaxSel Calls	ESS Calls	WeakDim Calls	Total	MaxSel	ESS	WeakDim	Total
Q01	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q08	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q09	256	400	20	676	0.0512	0.0800	0.0040	**0.1352**
Q10	32	400	20	452	0.0064	0.0800	0.0040	**0.0904**
Q11	256	20	1	277	0.0512	0.0040	0.0002	**0.0554**
Q12	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q13	64	160000	8000	168064	0.0128	32.0000	1.6000	**33.6128**
Q14	128	400	20	548	0.0256	0.0800	0.0040	**0.1096**
Q15	64	20	1	85	0.0128	0.0040	0.0002	**0.7722**
Q16	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q19	256	20	1	277	0.0512	0.0040	0.0002	**0.0554**
Q20	32	400	20	452	0.0064	0.0800	0.0040	**0.0904**
Q21	128	20	1	149	0.0256	0.0040	0.0002	**0.0298**
Q22	128	400	20	548	0.0256	0.0800	0.0040	**0.1096**
Q23	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q24	256	20	1	277	0.0512	0.0040	0.0002	**0.0554**
Q25	8	20	1	29	0.0016	0.0040	0.0002	**0.0058**
Q26	128	20	1	149	0.0256	0.0040	0.0002	**0.0298**
Q28	128	400	20	548	0.0256	0.0800	0.0040	**0.1096**
Q29	256	400	20	676	0.0512	0.0800	0.0040	**0.1352**
Q33	32	20	1	53	0.0064	0.0040	0.0002	**0.0106**

Table 6.8: `DimRed` TIME EFFICIENCY: TPC-DS (MSO MINIMIZATION)

Query	Overheads (Optimizer Calls)				Overheads (Time in Secs)			
Number	MaxSel Calls	ESS Calls	WeakDim Calls	Total	MaxSel	ESS	WeakDim	Total
Q03	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q07	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
Q12	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q15	32	8000	400	8432	0.0064	1.6000	0.0800	**0.0106**
Q18	256	3200000	160000	3360256	0.0512	640.0000	32.0000	**672.0512**
Q19	256	3200000	160000	3360256	0.0512	640.0000	32.0000	**672.0512**
Q21	128	8000	400	8528	0.0256	1.6000	0.0800	**1.7056**
Q22	16	8000	400	8416	0.0032	1.6000	0.0800	**1.6832**
Q26	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
Q27	512	160000	8000	168512	0.1024	32.0000	1.6000	**33.7024**
Q29	512	3200000	160000	3360512	0.1024	640.0000	32.0000	**672.1024**
Q36	16	8000	400	8416	0.0032	1.6000	0.0800	**1.6832**
Q37	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q40	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q42	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q43	8	400	20	428	0.0016	0.0800	0.0040	**0.0856**
Q52	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q53	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q55	32	400	20	452	0.0064	0.0800	0.0040	**0.0904**
Q62	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
Q63	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q67	16	8000	400	8416	0.0032	1.6000	0.0800	**1.6832**
Q73	32	8000	400	8432	0.0064	1.6000	0.0800	**0.0106**
Q82	512	8000	400	8912	0.1024	1.6000	0.0800	**1.7824**
Q84	256	3200000	160000	3360256	0.0512	640.0000	32.0000	**672.0512**
Q86	8	400	20	428	0.0016	0.0800	0.0040	**0.0856**
Q89	32	8000	400	8432	0.0064	1.6000	0.0800	**0.0106**
Q91	4096	3200000	160000	3364096	0.8192	640.0000	32.0000	**672.8192**
Q96	32	8000	400	8432	0.0064	1.6000	0.0800	**0.0106**
Q98	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q99	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**

6.8 Conclusions

Although, `SpillBound` class of algorithms bring welcome robustness guarantees, they are practical only for low-dimensional selectivity spaces since their compilation overheads are exponential in the

Table 6.9: `DimRed` TIME EFFICIENCY: JOB (MSO MINIMIZATION)

Query Number	Overheads (Optimizer Calls)				Overheads(Time in Secs)			
	MaxSel Calls	ESS Calls	WeakDim Calls	Total	MaxSel	ESS	WeakDim	Total
Q01	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q08	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q09	256	160000	8000	168256	0.0512	32.0000	1.6000	**33.6512**
Q10	32	400	20	452	0.0064	0.0800	0.0040	**0.0904**
Q11	256	8000	400	8656	0.0512	1.6000	0.0800	**1.7312**
Q12	16	20	1	37	0.0032	0.0040	0.0002	**0.0074**
Q13	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
Q14	128	8000	400	8528	0.0256	1.6000	0.0800	**1.7056**
Q15	32	8000	400	8432	0.0064	1.6000	0.0800	**1.6864**
Q16	16	400	20	436	0.0032	0.0800	0.0040	**0.0872**
Q19	256	160000	8000	168256	0.0512	32.0000	1.6000	**33.6512**
Q20	32	8000	400	8432	0.0064	1.6000	0.0800	**0.0106**
Q21	128	8000	400	8528	0.0256	1.6000	0.0800	**1.7056**
Q22	128	160000	8000	168128	0.0256	32.0000	1.6000	**33.6256**
Q23	32	160000	8000	168032	0.0064	32.0000	1.6000	**33.6064**
Q24	256	8000	400	8656	0.0512	1.6000	0.0800	**1.7312**
Q25	8	400	20	428	0.0016	0.0800	0.0040	**0.0856**
Q26	128	160000	8000	168128	0.0256	32.0000	1.6000	**33.6256**
Q28	128	160000	8000	168128	0.0256	32.0000	1.6000	**33.6256**
Q29	256	160000	8000	168256	0.0512	32.0000	1.6000	**33.6512**
Q33	32	8000	400	8432	0.0064	1.6000	0.0800	**0.0106**

dimensionality, and their performance bounds are quadratic in the dimensionality.

In order to address this limitation, we presented the `DimRed` pipeline, which leverages schematic, geometric and piggybacking techniques to reduce even queries with more than 15 dimensionality to five or less dimensions. In fact, for quite a number of queries, the dimensionality came down to the lowest possible value of 1! Gratifyingly, not only could we dramatically decrease the overheads due to such reductions, but could also significantly improve the quality of the performance guarantee.

Chapter 7

Reducing Overheads to Support Ad-Hoc Queries

7.1 Introduction

The `ESS` is computed from the dimensions remaining after explicit removal of dimensions by `SchematicRemoval` and `MaxSelRemoval` components of the `DimRed` pipeline, Even though `DimRed` provides substantial reduction in compilation overheads, a major limitation of `SpillBound` (SB) is that the reduced overheads are still manageable for canned queries but remain too high for ad-hoc queries.

An obvious first step towards addressing the above issue is to utilize multi-core computing platforms to leverage the intrinsic parallelism available in contour identification. However, this may not be sufficient to fully address the strong exponential dependence on dimensionality. In our view, adapting the SB methodology for ad-hoc queries requires, in addition to hardware support, *algorithmic approaches* for substantive reduction of the compilation overheads – the design of such approaches forms the focus of this chapter.

Problem Formulation   Specifically, we investigate the trade-off between the two key attributes of the SB approach, namely, the *compilation overheads* and the *MSO guarantee*. The overhead of SB is measured as the number of optimization calls made to the query optimizer in order to construct all the isocost contours. Given an algorithmic approach aimed at reducing this compilation overheads, we use γ (≥ 1) to denote its overheads reduction factor relative to the compilation overheads of SB. However, bringing down the compilation overheads may result in a weaker MSO guarantee. We use η (≥ 1) to denote this relaxation factor in the guarantee, relative to the MSO of SB. With this characterization, the formal problem addressed is the following:

Given a user query Q for which `SpillBound` *provides an MSO guarantee M, and a user-permitted relaxation factor η on this guarantee, design a query processing algorithm that maximizes γ while ensuring that the MSO guarantee remains within ηM.*

Algorithmic Reduction of the Overheads   The MSO guarantees of SB are only predicated on the standard assumption of *monotonic* behavior of plan cost functions with regard to `ESS` predicate selectivities. Here, we leverage the stronger fact that plan cost functions typically exhibit a *concave down* behavior in the `ESS` (and `PSS` as well) – i.e. they have monotonically non-increasing slopes.[1] Specifically, we design a modified algorithm, `FrugalSpillBound` (`FSB`), that incorporates the concave behavior to substantially reduce the compilation overheads at the cost of a mild relaxation on the MSO guarantee. Quantitatively, for `ESS` resolution r, the attractive tradeoff between η and γ is the following:

$$
\begin{aligned}
\gamma &= r/\log_\eta r & D &= 1 \\
\gamma &= \Omega(r^D/(D\log_\eta r)^{D-1}) & D &\geq 2
\end{aligned}
\tag{7.1}
$$

That is, the initial regime of `FSB` provides an *exponential improvement* in γ for a linear increase in η.

More concretely, a sample instance of the $\eta - \gamma$ tradeoff is shown in the red line of Figure 7.1, obtained for $r = 100$ (corresponding to selectivity characterization at 1% intervals) and a 4D `ESS` derived from Query 26 of the TPC-DS benchmark. In this figure, which graphs a *semi-log* plot, the initial exponential overhead reduction regime is long enough that a **two orders of magnitude improvement** in γ is achieved with an η of 2. Further, when *empirically* evaluated, the decrease in overheads is much greater – this is shown in the blue line of Figure 7.1, where **nearly four orders of magnitude improvement** in γ is achieved for $\eta = 2$.

The concavity assumption directly leads to an elegant `FSB` construction for the base case of a one-dimensional `ESS`. To handle the multi-dimensional scenario, however, we need additional machinery, called *bounded contour-covering sets* (BCS) – these sets serve as low-overhead replacements for the original isocost contours. More precisely, a BCS is a set of locations that collectively *spatially dominate* all locations on the associated contour, and whose costs are within a bounded factor of the contour cost. Efficient identification of the BCS is made possible thanks to the concavity assumption, and the aggregate cardinality of the BCS over the contours is *exponentially smaller* than the number of locations in the `ESS`, resulting in the substantially decreased overheads.

Performance Results   The empirical performance results indicate that a two orders of magnitude theoretical reduction in overheads is *routine* with $\eta = 2$, while the empirical reduction in overheads

---

[1] As explained in Section 7.2, a weaker form of concavity, called Axis-Parallel Concavity, is sufficient for our techniques to hold.
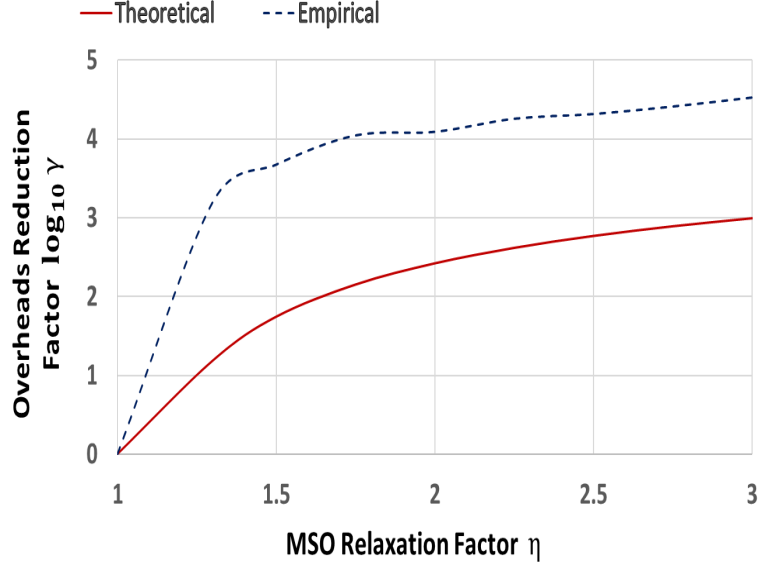
Figure 7.1: FSB $\eta - \gamma$ Tradeoff for 4D_Q26

is typically an order of magnitude more than this guaranteed value, delivering a cumulative benefit of more than three orders of magnitude. Therefore, the new FSB approach represents a substantive step towards practically achieving robust query processing for ad-hoc queries with moderate ESS dimensionalities – especially in conjunction with contemporary multi-core architectures that exploit the inherent parallelism in the ESS construction. So, for instance, a 5D query which takes a **few days** even on a well-provisioned multi-core machine to complete nearly 10 billion optimizer calls required for constructing the entire ESS (at a resolution of 100), can now be made ready for execution within a **few minutes** by FrugalSpillBound!

7.2 Assumptions

We augment the PCM assumption with a stricter condition, wherein not only are the PCFs monotonic, but also exhibit a weak form of *concavity* in their cost trajectories, as explained next.

7.2.1 Axis-Parallel Concavity (APC)

We augment the above PCM assumption with a stricter condition, wherein the PCFs are not only monotonic, but also exhibit a weak form of *concavity* in their cost trajectories, as explained next.

In the 1D world, a plan cost function $\mathcal{F}_p$ is said to be concave if, for any pair of locations q_1, q_2 in the 1D ESS, and any $\alpha \in [0, 1]$,

$$\mathcal{F}_p((1-\alpha)q_1 + \alpha q_2) \geq (1-\alpha)\mathcal{F}_p(q_1) + \alpha\mathcal{F}_p(q_2) \tag{7.2}$$

100

Generalizing to D dimensions, a PCF $\mathcal{F}_p$ is said to be *axis-parallel concave* (APC) if the function is concave along every axis-parallel 1D segment of the ESS. That is, Equation 7.2 is satisfied by any generic pair of locations q_1, q_2 in the ESS that belong to a common 1D segment of the ESS (i.e., $\exists j$ s.t. $q_1.k = q_2.k, \forall k \neq j$).

So, for example, if e_1 and e_2 are the epps of a 2D ESS, then the APC requirement is that each PCF should be concave along every vertical and horizontal line in the ESS.

Note that APC is a strictly *weaker* condition than complete concavity across all dimensions – that is, all fully-concave functions satisfy APC, but the reverse may not be true. Further, an important and easily provable implication of the PCFs exhibiting APC is that the corresponding OCS, which is the infimum of the PCFs, *also satisfies* APC. Finally, for ease of presentation, we will generically use concavity to mean APC in the remainder of this chapter.

Empirical Validation of APC

An immediate question that arises in the above context is whether the concavity assumptions on the PCFs (and, by implication, the OCS) generally hold true in practice. For this purpose, we have carried out extensive experimental evaluation with the TPC decision-support benchmarks operating on contemporary database engines. The summary finding of this empirical evaluation, whose details are presented later in Section 7.6, is that APC is consistently observed over almost the entire ESS.
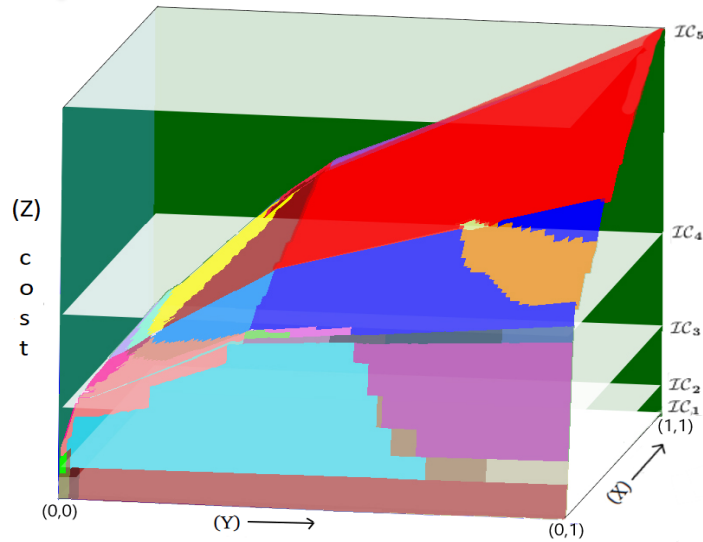

Figure 7.2: Optimal Cost Surface (OCS)

As a sample instance, the axis-parallel projections of a 3D OCS presented in Figure 7.2, are computed in Figures 7.3 (a) and 7.3 (b), respectively. These figures are graphed on a *log-log scale* and for ease of representation, capture only the optimality region of each PCF. We observe here that

the PCFs clearly exhibit concavity in their optimality regions with respect to selectivity. As a direct consequence, the OCS exhibits concavity over the entire selectivity range, justifying the assumption on which our results are based. Moreover, given current relational operator cost functions, a detailed rationale as to why PCF and OCS concavity is to be expected, is discussed below.

(a) Projection on X dimension   (b) Projection on Y dimension
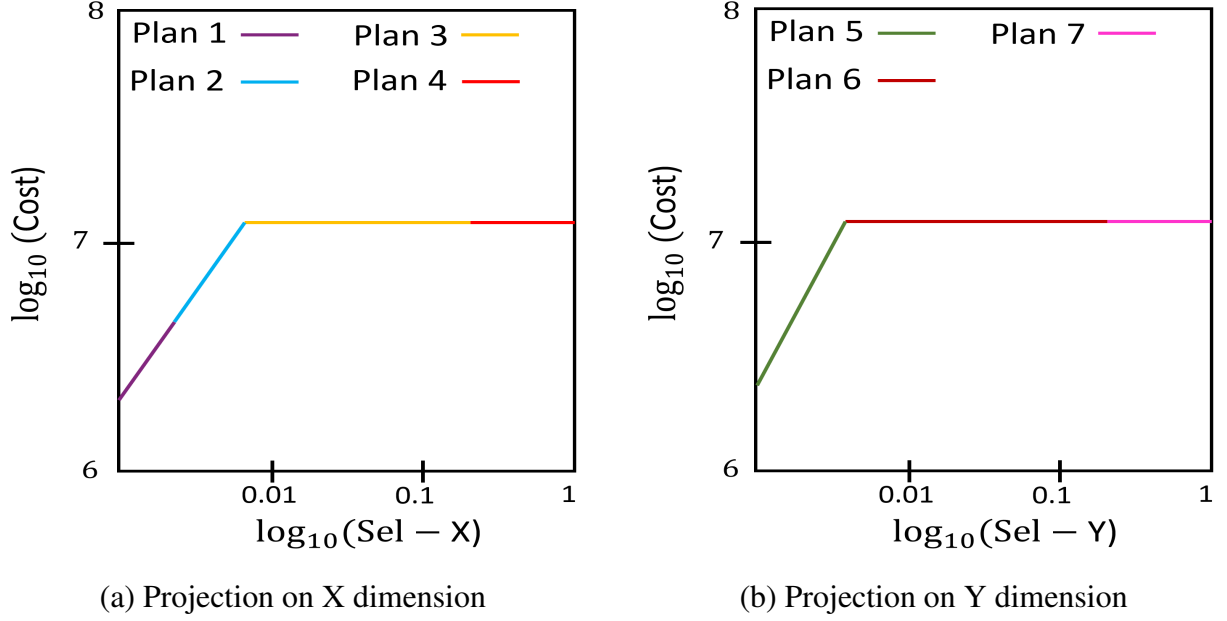
Figure 7.3: Validation of Axis-Parallel Concavity

Rationale of APC   Let us now see the rationale behind why we can expect axis-parallel concave behaviour of PCFs. Since a plan is a tree of operators, the individual operator cost behaviours are as follows: Except for the sort operator, we observed that all the operators are having non-increasing slope, piecewise linear or simply linear behaviour.

Starting with join operators, Hash Join and Merge Join (in case of already sorted inputs) costs $\mathcal{O}(s_x + s_y)$ where s_x, s_y represents the two input selectivities from its upstream operators. As we are interested in 1D axis-parallel projections, wherein one of s_x or s_y is constant, we can expect the two operators to behave linearly. Similar is the case with Nested Loop Join in which case it costs $\mathcal{O}(s_x * s_y)$. Shifting the focus to scanning operators, Sequential Scan has constant cost, whereas Index Scan and Bitmap Scan is linear. Finally, Sort operator has cost of the form $\mathcal{O}(s_x \log(s_x))$ which is superlinear. We can expect axis-parallel concave behaviours of PCFs, based on the following observations:

1. Estimated cost of a complete plan tree is an aggregate sum of costs of all its internal nodes - this appears to be case in current engines, especially, PostgreSQL.

2. In practice, contribution from Sort is very less compared to the total cost of a plan in industrial strength benchmarks. [DNC17]

3. Point-wise sum of a set of concave functions is also a concave function.

OCS concavity follows from the fact that OCS is a infimum of all POSP PCFs. Further, note that for the working of the `FrugalSpillBound` algorithm, we just need OCS to be axis-parallel concave.

7.3 Frugal SpillBound for 1D ESS

1D SB   We begin by reviewing how the `SpillBound` algorithm operates on a 1D `ESS` along with setting up the required notations to aid `FSB`'s description. Consider the sample concave OCS function $\mathcal{F}$ shown in Figure 7.4. In this figure, the selectivity axis represents the selectivity range for the lone `epp`, and the cost axis represents the OCS function. The cost axis is discretized into doubling-based isocost contours, $\mathcal{IC}_1$ through $\mathcal{IC}_m$, with $\mathtt{CC_i} = 2^{i-1}C$. Note that, in the case of 1D OCS, each of the contours correspond to a *single* selectivity location on the selectivity axis. We denote the location corresponding to $\mathcal{IC}_i$ by Q_i. Further, $Q_1 = 1/r$ and $Q_m = 1$ correspond to the origin and terminus locations, respectively.
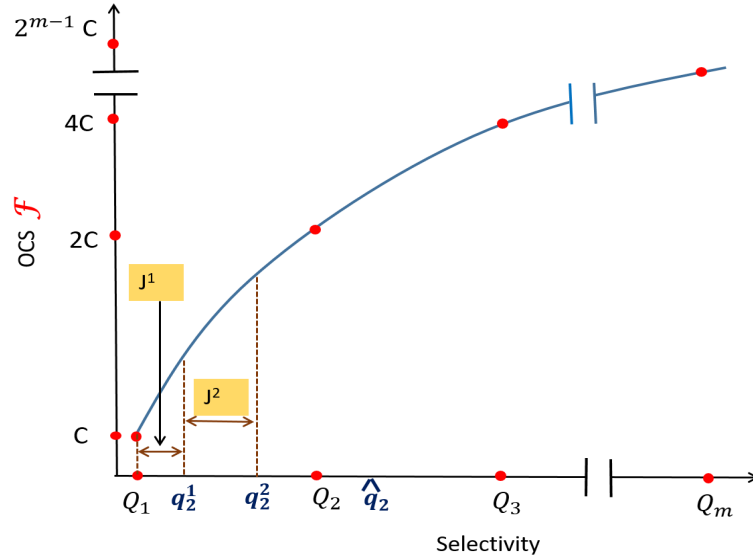
Figure 7.4: Concave OCS

During the compilation phase of `1D-SB`, for each of the r uniformly spaced locations on the selectivity axis, the optimal plan at the location and its cost are determined. Using the cost information from the r locations, the precise location of Q_i is identified for $i = 1, \ldots, m$. The set of optimal plans at the Q_i locations is called the "bouquet of plans". Then, during the execution phase, this

bouquet of plans is sequentially executed, starting from the cheapest isocost contour, with a budget equal to the associated contour cost. The process ends when a plan reaches completion within its allocated budget. This budgeted sequence of plan executions achieves an MSO guarantee of 4 with a compilation overhead of r optimizer calls.

We now move on to presenting our 1D-FSB algorithm, which has compilation and execution phases, as described below.

7.3.1 Compilation Phase

The main idea in the compilation phase of FSB is to dispense with SB's approach of precisely identifying the location of the Q_is. Instead, for each Q_i, we identify a *proxy location*, $\hat{q}_i$, such that the cost of the optimal plan at $\hat{q}_i$ is in the range $[\mathcal{F}(Q_i), \eta\mathcal{F}(Q_i)]$. The compilation phase consists of identifying these proxy locations $\hat{q}_i$s via a sequence of calibrated *jumps* in the selectivity space. Let us now see how to find these proxy locations starting with Q_2.

Discovering the proxy for Q_2

Since Q_1 is known, we set $\hat{q}_1 = Q_1$. The search for $\hat{q}_2$ starts from $\hat{q}_1$. We now perform a sequence of jumps in the selectivity space until we land exactly at Q_2 or overshoot it for the first time. Further, the lengths of the jumps are calibrated such that when $\hat{q}_2$ is reached, its cost is guaranteed to be in the range $[\mathcal{F}(Q_2), \eta\mathcal{F}(Q_2)]$, as described below.

First Jump   Identify the optimal plan $P_{\hat{q}_1}$ at $\hat{q}_1$, and compute its slope, $s(\hat{q}_1)$ at $\hat{q}_1$. Due to PCM, the slope at any location in $\mathcal{F}$ is > 0. The slope is calculated through abstract plan costing (a feature detailed in Section 8.1) of $P_{\hat{q}_1}$ at a selectivity location in the close neighborhood of $\hat{q}_1$.

Our first estimate for $\hat{q}_2$, denoted by q_2^1 (refer to Figure 7.4), is the location that is expected to have η times the cost of $\mathcal{F}(\hat{q}_1)$ when extrapolated by a tangent line with a slope of $s(\hat{q}_1)$, i.e,

$$\frac{\mathcal{F}(\hat{q}_1) + s(\hat{q}_1) \cdot (q_2^1 - \hat{q}_1)}{\mathcal{F}(\hat{q}_1)} = \eta$$

By rearranging, we get

$$q_2^1 = \hat{q}_1 + \frac{(\eta - 1) \cdot \mathcal{F}(\hat{q}_1)}{s(\hat{q}_1)}$$

$$\therefore q_2^1 = \hat{q}_1 + J^1$$

where J^1 represents the first jump towards $\hat{q}_2$, relative to the starting location, $\hat{q}_1$. The following lemma immediately follows from the concavity of the PCF:

Lemma 7.1 *The cost condition $\mathcal{F}(q_2^1) \leq \eta \cdot \mathcal{F}(\hat{q}_1)$ is satisfied.*

We next show that the jump J^1 is such that the selectivity of q_2^1 is at least η times the selectivity at $\hat{q}_1$.

Lemma 7.2 *The selectivity of q_2^1 is at least η times the selectivity at $\hat{q}_1$, i.e, $\boldsymbol{q_2^1 \geq \eta \hat{q}_1}$.*

Proof: Let the tangent of $\mathcal{F}$ at $\hat{q}_1$ be expressed as

$$\mathcal{F}(q) = s(\hat{q}_1) \cdot q + c' \qquad 0 \leq q \leq 1, c' \geq 0$$

(Here, $c' \geq 0$ to ensure non-negative cost at $q = 0$.) Based on this equation, we obtain the following pair of equations by separately considering the PCF cost at $\hat{q}_1$ and the estimated cost at q_2^1 .

$$\mathcal{F}(\hat{q}_1) = s(\hat{q}_1) \cdot \hat{q}_1 + c'$$
$$\eta \cdot \mathcal{F}(\hat{q}_1) = s(\hat{q}_1) \cdot q_2^1 + c'$$

Simplifying the equation pair, we get

$$\eta s(\hat{q}_1) \cdot \hat{q}_1 + \eta c' = s(\hat{q}_1) \cdot q_2^1 + c'$$
$$\therefore q_2^1 = \eta \hat{q}_1 + \frac{(\eta - 1)c'}{s(\hat{q}_1)} \geq \boldsymbol{\eta \hat{q}_1}$$

$\square$

Depending on the cost at the first jump's landing location, i.e. at q_2^1, two cases are possible:

1. *Cost Overshoot*, i.e, $\mathcal{F}(q_2^1) \geq \mathcal{F}(Q_2)$: In this case, we have identified a proxy location for Q_2 whose cost is at most $\eta \mathcal{F}(Q_2)$ (by Lemma 7.1).

2. *Cost Undershoot*, i.e., $\mathcal{F}(q_2^1) < \mathcal{F}(Q_2)$: In this case, the jump scheme is repeated with q_2^1 as the starting location. That is, we jump to q_2^2, with the jump length being $J^2 = \dfrac{(\eta - 1)\mathcal{F}(q_2^1)}{s(q_2^1)}$. This process is repeated until we reach $\hat{q}_2$, signalled by $\mathcal{F}(\hat{q}_2) \geq \mathcal{F}(Q_2)$. Since the cost of $\hat{q}_2$'s previous location is less than $\mathcal{F}(Q_2)$, Lemma 7.1 guarantees that $\mathcal{F}(\hat{q}_2) \leq \eta \mathcal{F}(Q_2)$.

7.3.1.1 Implementation of Proxy Discovery

The above compilation phase of FSB for the 1D scenario is detailed in Algorithm 3. Here, the entire search from $\hat{q}_1$ to $\hat{q}_2$ is captured as a generic Explore subroutine, with three arguments: seed, the starting location, t_cost, the cost at the terminal location, and r_factor, the relaxation factor wrt t_cost.

The proxy location $\hat{q}_i$ for Q_i is obtained starting with the proxy location $\hat{q}_{i-1}$. This is done by calling the Explore subroutine, with seed as $\hat{q}_{i-1}$, target cost of CC$_i$, and relaxation factor of η. The

derivation that bounded the relative cost of $\hat{q}_2$ w.r.t. the cost of Q_2 can be repeated to show that the cost of $\hat{q}_i$ is at most $\eta \mathcal{F}(Q_i)$ for $i = 2, \ldots, m-1$. Finally, the output of the algorithm is a set of proxy locations, ProxyContourLocs = $\{Q_1 \cup \{\bigcup_{i=2}^{m-1} \hat{q}_i\} \cup Q_m\}$.

7.3.1.2 Bounded Compilation Overheads

Theorem 7.1 *The compilation overheads reduction, γ, of* 1D-FSB *is at least* $\dfrac{r}{\log_\eta r}$.

Proof: From Lemma 7.2, the maximum number of jumps is required when the selectivity estimation at each jump is exactly η times the selectivity of the previous location. Therefore, the total number of query optimizer calls is bounded as follows:

$$\text{Total Optimization Calls} \ \leq \log_\eta \frac{Q_m}{Q_1} \leq \log_\eta r$$

Thus, the compilation overheads reduce from r to $\log_\eta r$. □

7.3.2 Execution Phase

The execution phase of FSB, as shown in Algorithm 3, is the same as that of SB except that the plan bouquet now consists of the optimal plans at the proxy locations in ProxyContourLocs. We therefore easily derive the following theorem for *maintaining the η constraint*.

Theorem 7.2 *The MSO relaxation of* 1D-FSB *is at most η.*

Proof: From the compilation phase, we know that the cost of a proxy location $\hat{q}_i$ is at most η times the cost of Q_i. The bounded cost of each proxy location ensures that the sequence of execution costs for the 1D-FSB plan bouquet is $C, 2\eta C, 4\eta C, \ldots$ (as opposed to $C, 2C, 4C, \ldots$ for 1D-SB). Since the MSO of 1D-SB is 4, it follows that the MSO of 1D-FSB is bounded by 4η. □

7.4 Frugal SpillBound for 2D ESS

In this section, we present the extension of 1D-FSB to the 2D case. For ease of exposition, we refer to the two epps as x and y, respectively.

In the 1D ESS, each contour was a single point. However, in 2D, it is a continuous *curve* as shown in Figure 7.5. Therefore, the step of identifying the proxy locations for Q_is has to be generalized so as to *cover* an isocost contour $\mathcal{IC}_i$ with an appropriate set of proxy locations. We achieve this by finding a *bounded contour-covering set* (BCS) of locations for each contour $\mathcal{IC}_i$. The definition of these sets and their identification procedure are presented next.

Algorithm 3 `1D-FSB` (η)

 1: **Compilation Phase:**
 2: set $Q_1 = 1/r$ and $Q_m = 1$;
 3: set $k = 2$;
 4: set ProxyContourLocs = $\{Q_1, Q_m\}$;
 5: set $\hat{q}_1 = Q_1$;
 6: **while** $k < m - 1$ **do**
 7:     $\hat{q}_k$ = Explore($\hat{q}_{k-1}$, $\mathrm{CC_k}$, η);
 8:     Add $\hat{q}_k$ to ProxyContourLocs;
 9:     k++;
10: **end while**

11: **function** Explore(seed, t_cost, r_factor);
12: compute $cost = \mathcal{F}(\mathsf{seed})$ (using an optimizer call);
13: **while** $cost <$ t_cost **do**
14:     compute $slope$ at seed (using abstract plan costing);
15:     $next\_jump = (\mathsf{r\_factor} - 1) \cdot \dfrac{cost}{slope}$;
16:     seed $+ = next\_jump$;
17:     $cost = \mathcal{F}(\mathsf{seed})$;
18: **end while**
19: **return** seed;
20: **end function**

21: **Execution Phase:**
22: **for** q in ProxyContourLocs **do**
23:     Execute optimal plan P_q with budget $\mathcal{F}(q)$;
24:     **if** P_q completes execution **then**
25:         Return query result;
26:     **else**
27:         Terminate P_q and discard partial results;
28:     **end if**
29: **end for**

7.4.1 Bounded Contour-covering Set (BCS)

The BCS for a contour is defined as the set of locations such that:

(a) Every location in the contour is *spatially dominated* by at least one location in this set; and

(b) The cost of each location in BCS is *bounded* to within an η factor of the contour cost.

We denote the BCS of contour $\mathcal{IC}_i$ by BCS_i. Formally, BCS_i is a set that needs to satisfy the following condition:

$$\forall q \in \mathcal{IC}_i, \exists\, q' \in BCS_i \text{ such that } q \preceq q' \text{and } \text{COST}(q') \leq \eta \text{CC}_i$$

To make this notion concrete, a candidate BCS_i for the example contour $\mathcal{IC}_i$ shown in Figure 7.5, is $\{c_1, c_2, c_3\}$ which covers the entire contiguous length of the contour. As a specific case in point, the covering location c_2 fully covers the optimality segments of P_5 and P_6, as well as parts of P_4 and P_7, in $\mathcal{IC}_i$.
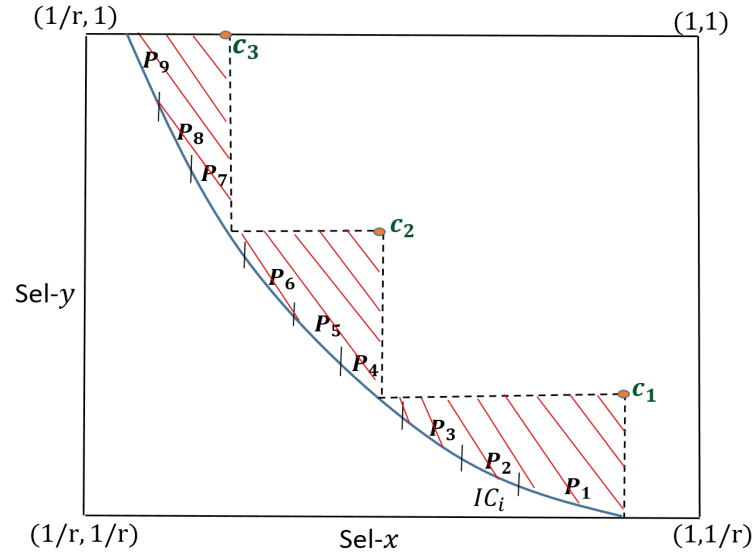
Figure 7.5: Bounded Contour-covering Set (BCS)

7.4.2 Compilation Phase

We now present a computationally efficient method to find a BCS for an isocost contour in the 2D ESS. To generalize the 1D method, we carry out jumps in the selectivity space along *both* the x and y dimensions. These jumps are designed to be axis-parallel and we leverage APC in their analysis. A special feature, however, is that the jumps are in *opposite* directions in the two dimensions – *forward* in

108

one, and *reverse* (i.e. jumps are performed in the decreasing selectivity direction) in the other. Further, in the reverse jumps, the selectivity of the next location is decreased by a *constant* factor, as explained below – this is in marked contrast to the forward jumps, where the Explore (seed,t_cost,r_factor) subroutine is invoked to decide the next location.

In principle, the choice of dimensions for forward and reverse jumps can be made arbitrarily. However, for ease of presentation, we assume hereafter that all forward jumps are in the y dimension, and all reverse jumps are in the x dimension.

7.4.2.1 Algorithm Description

We explain the compilation phase by describing the process of constructing BCS_i for the isocost contour $\mathcal{IC}_i$ shown in Figure 7.5. For ease of presentation, we refer to Figure 7.6, which overlays the construction of BCS_i on top of contour $\mathcal{IC}_i$.

The main idea is to carry out a sequence of interleaved search steps that alternatively explore the x and y dimensions. Specifically, we start from the location $c_0 = (1, 1/r)$ as the seed, and search for a location, u_1, on $y = 1/r$ line whose cost is in the range $[\texttt{CC}_\texttt{i}, \sqrt{\eta}\texttt{CC}_\texttt{i}]$. A sequence of reverse jumps from c_0, with constant $\sqrt{\eta}$ factor decrease in selectivity each time, is carried out until we reach u_1. The Explore subroutine is now invoked along the increasing y dimension with u_1 as the seed location, terminating cost $\sqrt{\eta}\texttt{CC}_\texttt{i}$, and relaxation factor $\sqrt{\eta}$. Let the location returned be c_1, and by the construction of Explore, we know that its cost is in the range $[\sqrt{\eta}\texttt{CC}_\texttt{i}, \eta\texttt{CC}_\texttt{i}]$. Now, starting from c_1, a sequence of reverse jumps, again with $\sqrt{\eta}$ factor selectivity decrease in each jump, is carried out till we reach a location u_2 whose cost is in the range $[\texttt{CC}_\texttt{i}, \sqrt{\eta}\texttt{CC}_\texttt{i}]$. This is followed by a call to Explore with u_2 as the seed and the same settings as before for the other arguments. The returned location is now c_2. This interleaved process of reverse jumps along the x dimension and forward jumps along the y dimension, is repeated until the process hits the boundary of the ESS. Let us say that the process ends at location c_k ($k = 4$ for the example contour in Figure 7.6). Then, the set $BCS_i = \{c_1, \ldots, c_k\}$ is returned as the BCS of contour $\mathcal{IC}_i$. This description of the compilation phase of 2D-FSB is codified in Algorithm 4.

7.4.2.2 Proof of Correctness

In order to demonstrate that every location in the contour is spatially dominated by at least one location in the associated BCS, we need to first prove that reverse jumps allow us to find u_is, whose costs are in the range $[\texttt{CC}_\texttt{i}, \sqrt{\eta}\texttt{CC}_\texttt{i}]$. Equivalently, it is sufficient to show that each reverse jump results in a relative cost decrease of at most $\sqrt{\eta}$. To do so, let us fix a covering location c_k, and let $\mathcal{F}_{ap}$ denote the restriction of OCS to the horizontal line passing through c_k. Then, we have the following lemma:
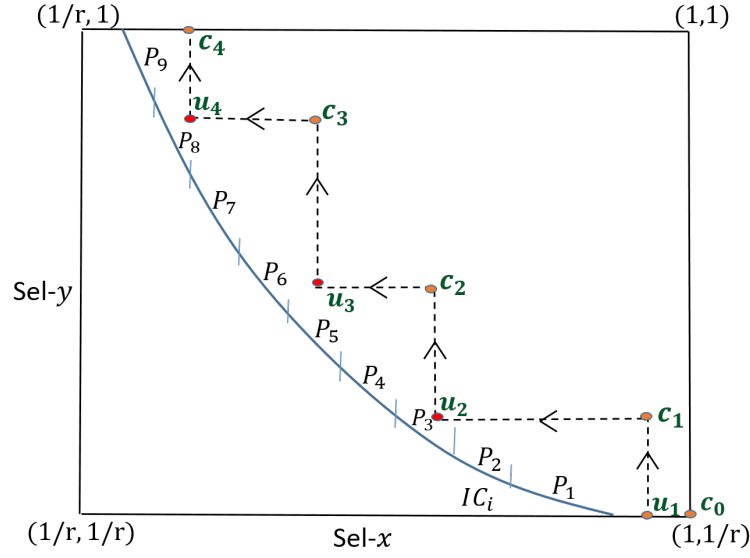
Figure 7.6: Identification of BCS

Lemma 7.3 *The reverse jump from a location q along the x direction by a factor $\sqrt{\eta}$ results in a relative cost decrease of at most $\sqrt{\eta}$, i.e., $\mathcal{F}_{ap}(\frac{q.x}{\sqrt{\eta}}, q.y) \geq \mathcal{F}_{ap}(q.x, q.y)/\sqrt{\eta}$.*

Proof: Let q' denote the location $(q.x/\sqrt{\eta}, q.y)$. Consider the line passing through q', parallel to the x-axis, and tangent to OCS. Let s be its slope and c' its intercept on the cost axis ($c' \geq 0$ to ensure non-negative cost at the origin). We know that this line *overestimates* the cost at q because $\mathcal{F}_{ap}$ is both increasing and concave (by virtue of its PCM and APC characteristics). Thus, we have

$$
\begin{aligned}
\mathcal{F}_{ap}(q) & \leq s \cdot (q.x) + c' \\
& \leq \sqrt{\eta}\left(s \cdot (\frac{q.x}{\sqrt{\eta}}) + \frac{c'}{\sqrt{\eta}}\right) \\
& \leq \sqrt{\eta}\left(s \cdot (\frac{q.x}{\sqrt{\eta}}) + c'\right) \\
& = \sqrt{\eta}\mathcal{F}_{ap}(q')
\end{aligned}
$$

where the second and third inequalities are implied by $\eta \geq 1$ and $c' \geq 0$. The last equality follows from the fact that the line passes through q'. □

Lemma 7.4 *Every location in $\mathcal{IC}_i$ is dominated by at least one location in BCS_i.*

Proof: Consider any point q in $\mathcal{IC}_i$. By construction we know that there exists $c_k \in BCS_i$ s.t. $c_k.y \leq q.y \leq c_{k+1}.y$. We will show that $c_{k+1} \in BCS_i$ is a dominating location for q by proving $q.x \leq c_{k+1}.x$. Consider the location u_{k+1} whose x coordinate is the same as that of c_{k+1}. This means

that (a) $u_{k+1}.x = c_{k+1}.x$, and (b) $u_{k+1}.y = c_k.y$. Since the cost of location u_{k+1} is $\geq$ the cost of location q, and $u_{k+1}.y \leq q.y$ by PCM, it implies that $u_{k+1}.x \geq q.x$. Therefore, q is dominated by c_{k+1}. $\qquad\square$

7.4.2.3 Bounded Computational Overheads

Now that we have shown the coverage properties of the BCS, we move on to proving that their identification can be accomplished with bounded overheads.

Lemma 7.5 *The overheads reduction, γ, of* 2D-FSB *is at least* $\dfrac{r^2}{4 \cdot m \cdot \log_\eta r}$.

Proof: Let us first consider the number of optimization calls required per contour for 2D-FSB. We know that the exploration of c_ks and u_ks move unidirectionally along the y-axis ($1/r$ to 1) and x-axis (1 to $1/r$), respectively. Furthermore, we have earlier shown that each jump results in a relative increase (or decrease) in selectivity of at least $\sqrt{\eta}$. Thus, by geometric progression, we can infer the following:

$$\text{Opt. calls per Contour} = \text{Opt. calls for } c_k\text{s} + \text{Opt. calls for } u_k\text{s}$$
$$\leq \log_{\sqrt{\eta}} r + \log_{\sqrt{\eta}} r$$
$$= 2\log_{\sqrt{\eta}} r = 4 \cdot \log_\eta r$$

Since there are m contours in the ESS, we conclude that there are $4 \cdot m \cdot \log_\eta r$ optimization calls across all contours for 2D-FSB, as compared to r^2 for 2D-SB. Thus, γ is at least $\dfrac{r^2}{4 \cdot m \cdot \log_\eta r}$. $\quad\square$

7.4.3 Execution Phase

In the execution-phase, we run the original 2D-SB algorithm, treating the BCS identified for every contour as the *effective contour*. Specifically, starting from the least cost BCS_1, the plans corresponding to the locations in each successive BCS_i are executed as per the 2D-SB algorithm. This BCS-based execution of plans (in spill-mode) is continued until the actual selectivities of both the epps are learned. Finally, the optimal plan at the discovered selectivity location is executed to completion to compute the query results for the user. We show below that with this execution strategy, the MSO guarantee is relaxed by at most η.

7.4.3.1 Maintaining the η constraint

Theorem 7.3 *The MSO relaxation of* 2D-FSB *is at most* η.

Proof: We know that the cost of any location in BCS_i is at most ηCC_i. Furthermore, the execution-phase runs the 2D SB algorithm on the BCS of every contour. Thus, every execution in 2D-FSB is

---

Algorithm 4 2D `FrugalSpillBound` Algorithm (η)

1: **Compilation Phase:**
2: Set: $q_{cur} = (1/r, 1/r)$;
3: Set: $\beta = \sqrt{\eta}$;
4: **while** contours are remaining **do**
5:     */*Let $\mathcal{IC}_i$ denote current contour and CC_i be its cost*/*
6:     **while** $q_{cur}.x \geq \frac{1}{r}$ and $q_{cur}.y \leq 1$ **do**
7:         Find u_i with cost in $[CC_i, \beta \cdot CC_i]$, by x-axis reverse jumps;
8:         $q_{cur}.x = u_i.x$;
9:         Call Explore($u_i, \beta \cdot CC_i, \beta$) along y-axis to find c_i;
10:         $q_{cur}.y = c_i.y$;
11:     **end while**
12:     Union of all c_is forms the bounded contour-covering set, BCS_i;
13:     */* Move to next contour */*
14: **end while**

15: **Execution Phase:**
16: Run the original 2D `SpillBound` algorithm on the plans corresponding to BCS_i, of every contour $\mathcal{IC}_i$;

---

performed with a budget of η times its corresponding contour cost. Hence, the overall cost of `2D-FSB` is at most η times that of 2D SB, which increases the MSO by at most η. $\qquad\square$

The analysis of the 2D `SpillBound` algorithm relied on two crucial properties: Half-Space Pruning (HSP) and Contour Density Independent Execution (CDIE) as described in Section 4.2. Both HSP and CDIE properties continue to hold for 2D `FSB` also, which is explained next.

7.4.3.2   Half-Space Pruning and Contour Density Independent Execution

With regard to identifying the set of plans to be executed and which `epp` to spill on, SB and `FSB` employ the same procedure which is sufficient to establish the CDIE property of `FSB`.

Moving to HSP, in SB, at any point in time, if a location q is being explored, it is always ensured that the selectivity value of q along non-`epp` dimensions is set to their selectivity at q_a. However, when we replace a continuous isocost contour with a discrete covering set in `FSB`, at the first sight it might look that the budget during partial executions may not be sufficient for HSP to hold true. But the key observation is that, when a location q is explored in `FSB`, for a non-`epp` dimension j, we ensure that $q.j \geq q_a.j$. This condition ensures that the budget is sufficient such that HSP (Lemma 4.1) applies to `FSB` as well.

7.4.3.3   Contour Covering Set identification

In the original SB algorithm, once a selectivity is completely learnt for an `epp`, the original `ESS` gets projected on the selectivity value of the learnt dimension. This process of reducing the dimensionality

112

of the `ESS` by 1 continues as and when an `epp` get completely learnt, until the actual selectivity of all the `epps` are discovered.

The isocost contours in the SB are continuous whereas the contour covering sets are discrete sets. Therefore, the update to the effective `ESS` by projecting the current `ESS` onto the selectivity of the learnt dimension needs to be done carefully in the case of `FSB`. The sensitive situation is when the learnt selectivity value is such that, there is no location in the covering set whose value on the learnt dimension is exactly equal to the learnt value. For example, in Figure 7.6, say that `FSB` learns the complete or actual selectivity in the Y-dimension first, whose value is strictly in between $u_3.y$ and $c_3.y$, i.e. $u_3.y < q_a.y < c_3.y$. In this case, we have to take care to ensure that, we project the 2D `ESS` onto the line $y = c_3.y$ to ensure that there is a valid starting locations for the 1D search along x dimension.

7.5 Multi-Dimensional FSB

In this section, we show how to extend `2D-FSB` to higher dimensions. For ease of exposition, we initially describe the algorithm and present the corresponding proof of correctness intuitively. After this, we move on to formalizing the intuitive proofs.

7.5.1 Multi-D Algorithm

The `MultiD-FSB` algorithm is executed on the set of dimensions retained after the DimRed pre-processing step. The retained `epps` are first ordered in decreasing value of their *inflation factors* (as defined in Section 6.3), i.e, e_1 has the highest inflation factor and e_D, the lowest. Then, for every contour, we construct a specially designed sparse grid $\mathcal{G}$ for the first $(D-2)$ dimensions. It is constructed such that there are totally $(\log_\beta(1/r))^{D-2}$ points, where $\beta = \sqrt[D]{\overline{\eta}}$. Further, in each dimension, there are $\log_\beta(1/r)$ points that are spread out in a geometric distribution with factor β. The key feature of this grid is that, even if we restrict the search space in the first $D-2$ dimensions to just the points in $\mathcal{G}$, we incur an MSO relaxation factor no more than β^{D-2} while still ensuring complete coverage of the underlying contour – the proof of this can be seen in Lemma 7.8.

The algorithm to cover an isocost contour $\mathcal{IC}_i$ runs in two important steps:

S1: Corresponding to each point p in the grid $\mathcal{G}$, run a `2D-FSB` with the first $(D-2)$ dimensions fixed as per p, and the last two dimensions at the full resolution of r. The output is treated as a subset of BCS_i.

S2: Compute the union of the `2D-FSB` outputs obtained in step **S1** over all the points in $\mathcal{G}$ – this union forms the final BCS_i.

Each invocation of `2D-FSB` incurs an MSO relaxation factor of β^2 corresponding to the last two dimensions. Further, $\mathcal{G}$ contributes an MSO relaxation factor of β^{D-2} due to the first $(D-2)$ dimensions. Thus, the overall MSO relaxation is contained at β^D, i.e, η. The pseudocode for `MultiD-FSB` is presented in Algorithm 5, and the proofs of overheads reduction and maintenance of the η relaxation constraint, can be seen in the following section.

Algorithm 5 `MultiD-FSB` (η)

1: **Compilation Phase:**
2: Set: $\beta = \sqrt[D]{\eta}$;
3: Set: $k = 1$; /*initialization to first contour*/
4: **while** contours are remaining **do**
5:     Set: $q_{cur} = (\frac{1}{r}, \cdots, \frac{1}{r})$; /*starting to explore the kth contour*/
6:     $BCS_k = \emptyset$
7:     **for** $q_{cur}.1 = \frac{1}{r}$; $q_{cur}.1 \leq 1$; $q_{cur}.1 = \beta q_{cur}.1$ **do**
8:         /* $D - 3$ *more nested* **for** *loops like the above corresponding to the dimensions* 2 *through* $(D-2)$*/
9:         /* At the end of $(D-2)$ nested **for** loops, q_{cur} is such that its first $(D-2)$ dimensions correspond to one of the points in the special grid $\mathcal{G}$ */
10:         $q_{min} = q_{max} = q_{cur}$;
11:         $q_{max}.(D-1) = q_{max}.D = 1$;
12:         $q_{min}.(D-1) = q_{min}.D = \frac{1}{r}$;
13:         /*q_{min} *and* q_{max} *are origin and terminus of 2D space of dimensions* $(D-1)$ *and* D*/
14:         **if** $Cost(q_{min}) \leq \text{CC}_k$ and $Cost(q_{max}) \geq \text{CC}_k$ **then**
15:             Augment BCS_k with the output of `2D-FSB` covering a 2D contour of cost $(\beta)^{D-2}\text{CC}_k$ with cost relaxation factor of β^2;
16:         **end if**
17:     **end for**   /* *End of* $(D-2)$ *nested* **for** *loops* */
18:     Output BCS_k and set $k = k + 1$; /* *Move to next contour* */
19: **end while**

20: **Execution Phase:**
21: Run the multi-D `SpillBound` algorithm on the plans corresponding to BCS_i for each contour $\mathcal{IC}_i$;

7.5.2   Proof of Correctness

We shall begin with introducing the notion of a *sub-contour*, denoted by $\mathcal{IC}|_E^H$, for any contour $\mathcal{IC}$. It is defined to be the set of locations that belong to intersection of *hyperplane H* and $\mathcal{IC}$, and then projected on the dimensions $E \subseteq \text{EPP}$. Note that any location $q \in \mathcal{IC}|_E^H$ would be a $|E|$ dimensional location. Furthermore, a sub-contour, $\mathcal{IC}_{i_1}|_E^H$, *covers* another sub-contour $\mathcal{IC}_{i_2}|_E^H$, if $\nexists$ a location $q \in \mathcal{IC}_{i_2}|_E^H$ s.t. q dominates[1] some location of $\mathcal{IC}_{i_1}|_E^H$. In words, there should not exist a location in $\mathcal{IC}_{i_2}|_E^H$ which dominates some location in $\mathcal{IC}_{i_1}|_E^H$. Furthermore, for any location $q \in \text{ESS}$, we use

---

[1]By domination, we mean strict domination

$q|_E$ to denote the projection of q on dimensions E. Notation $[n], n \in \mathcal{Z}^+$ is used represent the set of integers from $\{1, \cdots, n\}$. Finally, we can also represent a location $q \in$ ESS by its direct sum of its hyperplanes, $q_1|_{[D-2]}$ and $q_2|_{D-1,D}$ for some $q_1, q_2 \in$ ESS. Denoting the direct sum by $\circ$, then

$$\{q := q_1|_{[D-2]} \circ q_2|_{D-1,D}\} \Leftrightarrow \{q.j = q_1.j, j \in [D-2] \text{ and } q.j = q_2.j, j \in \{D-1, D\}\}$$

Lemma 7.6 *For some two locations $q_1, q_2 \in \mathcal{IC}$ and $q_2|_{[D-2]} \succ q_1|_{[D-2]}$, the 2D contour obtained by intersecting the hyperplane $q_1|_{[D-2]}$ with $\mathcal{IC}$, covers the one obtained by intersecting $\mathcal{IC}$ with $q_2|_{[D-2]}$. That is, sub-contour $\mathcal{IC}|_{D-1,D}^{q_1|_{[D-2]}}$ covers $\mathcal{IC}|_{D-1,D}^{q_2|_{[D-2]}}$.*

Proof: For sake of contradiction, let us say that sub-contour $\mathcal{IC}|_{D-1,D}^{q_1|_{[D-2]}}$ *does not* cover $\mathcal{IC}|_{D-1,D}^{q_2|_{[D-2]}}$. Further, this means there exist a 2D location $q_2' \in \mathcal{IC}|_{D-1,D}^{q_2|_{[D-2]}}$ which dominates some 2D location, $q_1' \in \mathcal{IC}|_{D-1,D}^{q_1|_{[D-2]}}$. Since $q_2|_{[D-2]} \succ q_1|_{[D-2]}$, we can conclude

$$\{q_2 := q_2|_{[D-2]} \circ q_2'\} \succ \{q_1 := q_1|_{[D-2]} \circ q_1'\} \tag{7.3}$$

By PCM, there is a contradiction that q_1, q_2 belong to the same contour and have same cost. $\quad\square$

Lemma 7.7 *Consider two distinct contours $\mathcal{IC}_1, \mathcal{IC}_2$ s.t. $\mathtt{CC}_2 = (\beta)^{D-2} * \mathtt{CC}_1$, and two hyperplanes defined by $D-2$ dimensional locations, q and q_β where $q_\beta.j = \beta * q.j, \ \forall j \in [D-2]$. Then, the 2D contour obtained by intersecting q_β hyperplane with $\mathcal{IC}_2$, covers the one obtained by intersecting q hyperplane with $\mathcal{IC}_1$. That is, sub-contour $\mathcal{IC}_{i_2}|_{D-1,D}^{q_\beta}$ covers $\mathcal{IC}_{i_1}|_{D-1,D}^{q}$.*

Proof: For sake of contradiction, let us say that sub-contour $\mathcal{IC}_{i_2}|_{D-1,D}^{q_\beta}$ *does not* cover $\mathcal{IC}_{i_1}|_{D-1,D}^{q}$. Further, this means there exist a 2D location $q_1' \in \mathcal{IC}_{i_1}|_{D-1,D}^{q}$ which dominates some 2D location, $q_2' \in \mathcal{IC}_{i_2}|_{D-1,D}^{q_\beta}$. We know that location $q' \in \mathcal{IC}_1$, where $q' := q \circ q_1'$. This means that cost of location q' is equal to $\mathtt{CC}_1$. Since $q_1' \succ q_2'$, cost of location q_3', where $q_3' := q \circ q_2'$, is strictly less than $\mathtt{CC}_1$. By concavity, then the cost of the location q_β', $q_\beta' := q_\beta \circ q_2'$, has cost strictly less than $\mathtt{CC}_2$. This contradicts the fact that $q_\beta' \in \mathcal{IC}_2$, and thus having cost equal to $\mathtt{CC}_2$. $\quad\square$

Lemma 7.8 *Let BCS_i be the bounded contour-covering set output for contour $\mathcal{IC}_i$ using Multi-D* FSB*, then*

1. *every location in $\mathcal{IC}_i$ is dominated by some location in BCS_i*

2. *cost of the dominating location (found in part 1) in BCS_i is at most $\eta \cdot \mathtt{CC_i}$*

Proof:

1. Consider a location $q \in \mathcal{IC}_i$. The proof goes by eventually constructing a location $dom \in BCS_i$, which dominates q. We know that each of $\{q_{cur}.1, \cdots, q_{cur}.(D-2)\}$ in Algorithm 5, is iteratively increased from $1/r \to 1$, with a step size of β. Thus, it is easy to see that there exist a q_{cur} such that

$$\frac{q_{cur}.j}{\beta} < q.j \leq q_{cur}.j \ \forall j \in [D-2]$$

   In other words, q_{cur} dominates q when projected on its first $D-2$ dimensions. and thus we set $dom.j = q_{cur}.j \ \forall j \in [D-2]$. Let q_β be D dimensional location such that $q_\beta.j = q_{cur}.j/\beta$. Then, Lemma 7.6 shows that sub-contour $\mathcal{IC}_i|_{D-1,D}^{q_\beta|[D-2]}$ covers $\mathcal{IC}_i|_{D-1,D}^{q|[D-2]}$. Furthermore from Lemma 7.7, we know that sub-contour $\mathcal{IC}_{i_1}|_{D-1,D}^{q_{run}|[D-2]}$ covers $\mathcal{IC}_i|_{D-1,D}^{q_\beta|[D-2]}$ where $\text{CC}_{i_1} = (\beta)^{D-2} * \text{CC}_i$. From 2D FSB, we know that every location in $\mathcal{IC}_{i_1}$ is covered by the covering set. This implies that there exist an assignment of values for $(dom.(D-1), dom.(D))$ s.t. dom dominates q.

2. Since we are covering the $(\beta)^{D-2} \cdot \text{CC}_i$ contour, with an cost inflation factor of at most β^2. Leveraging the proof of Lemma 7.4, we conclude that any location in BCS_i has at most $\eta \cdot \text{CC}_i$ cost.

$\square$

Next, let us bound the number of optimization calls required by multi-D FSB algorithm per contour.

Lemma 7.9 *The number of optimization calls per contour, for an MSO relaxation of η, is upper bounded by* $2 * (D * \log_\eta(r))^{D-1}$

Proof: We know from Lemma 7.5 that total number of optimization calls for 2D FSB is $2 * \log_\beta(r)$. However, the 2D algorithm is executed $(\log_\beta(r))^{D-2}$ times. Equivalently the factor can be rewritten as, $(\frac{\log_2(r)}{\log_2(\sqrt[D]{\eta})})^{D-2}$, or $\{D^{D-2} * (\log_\eta(r))^{D-2}\}$. Thus, optimization calls per contour is upper bounded by $2 * (D * \log_\eta(r))^{D-1}$. $\square$

Theorem 7.4 *The compile-time overheads reduction, γ, of Multi-D FSB is at least $r^D/(2 \cdot m \cdot (D \cdot \log_\eta r)^{D-1})$.*

Theorem 7.5 *MSO relaxation of Multi-D FSB is at most η.*

Proof: Using Part 2 of Lemma 7.8 and analysis similar to that of 2D scenario, the proof follows. $\square$

7.6 Experimental Evaluation

In this section, we profile the $\gamma - \eta$ performance of FSB using SB's performance as the reference baseline. For ease of exposition, and that the overheads for queries with less than three dimensions are in few hundreds of optimizer calls, we present the results for queries with three or more dimensions.

7.6.1 Empirical Validation of APC

We begin with an experimental validation of the APC assumption that is central to the FSB approach. For this purpose, we obtained the cost functions of the POSP plans over the ESS using the selectivity injection feature for all the queries in our evaluation suite. Then, we verified, for each cost function, whether its slope was monotonically non-increasing with selectivity for every 1D projection of the function. Representative results of this evaluation, reflecting 120-plus plans sourced from our query workload, are tabulated in Table 7.1, for both the constituent PCFs and the aggregate OCS.

In the table, a cell corresponding to OCS (or PCF), under *Average*, captures the % of locations in ESS satisfying the assumption averaged over OCSs (or PCFs) in a query along different projections. Supporting metrics such as *Median*, *Minimum* and *Maximum* are also enumerated to provide a sense of the overall distribution. Note that our FSB approach requires concavity only on the OCS, and the vast majority ($> 95\%$) of locations in the ESS satisfy this slope constraint. Moreover, the median value being 100% for most queries indicates that the majority of OCSs and PCFs do not violate the assumption at all. Further, even the rare violations that surfaced were found to be artifacts of rounding errors, cost-modeling errors, and occasional PCM violations due to the PostgreSQL query optimizer not being entirely cost-based in character.

7.6.2 Theoretical Characterization of $\gamma - \eta$

Using the formula derived in Theorem 7.4, we evaluated the γ value for our suite of benchmark queries with η set to 2, and these results are shown in Figure 7.7 on a *log scale*. We observe a consistent overheads decrease by more than two orders of magnitude for FSB, i.e. $\gamma >= 100$, over all the queries. Further, the decrease shows a trend of being *magnified* with dimensionality – for instance, the overheads decrease by a factor of almost 400 for the five-dimensional 5D_Q84.

7.6.3 Empirical Characterization of $\gamma - \eta$

We now turn our attention to assessing the *empirical* reduction in compilation overheads achieved by FSB for the above database environment – these results are also captured in Figure 7.7. We see here that for most of the queries, the savings are over *three* orders of magnitude. Furthermore, quite a few of the 4D and 5D queries even reach *four* orders of magnitude reduction – in fact, the overheads saving for 5D_Q91 is by a factor of almost 40000! When the effective dimensionality and the number

Table 7.1: % LOCATIONS IN ESS SATISFYING APC

Query		Average	Median	Min.	Max.
3D_Q15	OCS	100	100	100	100
	PCF	100	100	100	100
3D_Q96	OCS	100	100	100	100
	PCF	100	100	100	100
4D_Q7	OCS	100	100	100	100
	PCF	98.4	100	74.4	100
4D_Q26	OCS	99.7	100	98.8	100
	PCF	99.7	100	93.9	100
4D_Q27	OCS	100	100	100	100
	PCF	99.2	100	75.2	100
5D_Q19	OCS	100	100	100	100
	PCF	100	100	100	100
5D_Q84	OCS	96.9	96.5	96.5	97.6
	PCF	94.5	96.8	71.2	100
5D_Q91	OCS	100	100	100	100
	PCF	100	100	98.4	100

of contours is moderate, as in the case of few 3D queries in Figure 7.7, the savings become saturated at around 2.5 orders of magnitude since the overheads reach a low value in absolute terms itself, of the order of a few thousand optimization calls.

The reasons for the considerable gap between the theoretical and empirical values include the following:

- Our conservative formulation in Lemma 7.2 for the distances covered by the forward jumps in FSB. These jumps are based on the slope of the optimal plan function at the corresponding location, but the lengths of the jumps in practice are considerably more due to the concave trajectory. For instance, we found that with 5D_Q84, around 60 percent of the jump lengths exceeded 1.5 times the guaranteed value, while about 20 percent were more than twice the guaranteed value.

- Our conservative assumption that all covering contours start from $1/r$ and work their way upto the maximum selectivity of 1. In practice, however, the contour traversals could be much shorter. As a case in point, we found that with 5D_Q84, around 80 percent of the underlying 2D contour explorations were skipped based on the cost condition check in Line 14 of Algorithm 5.
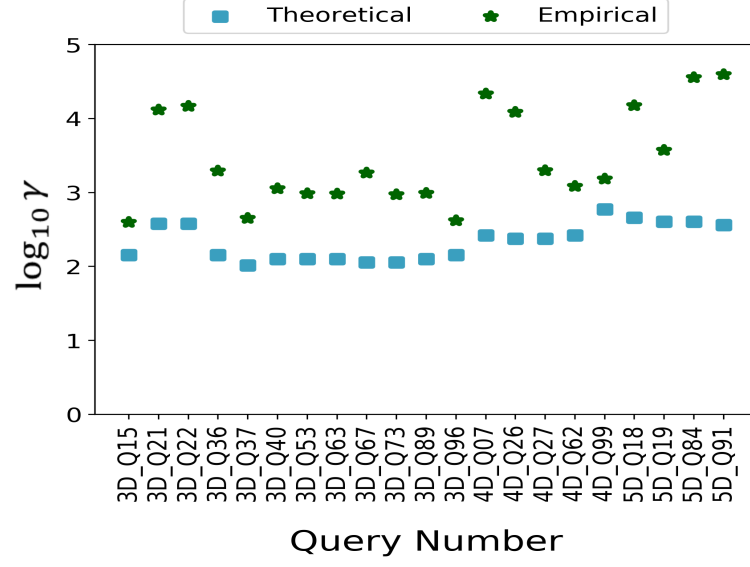
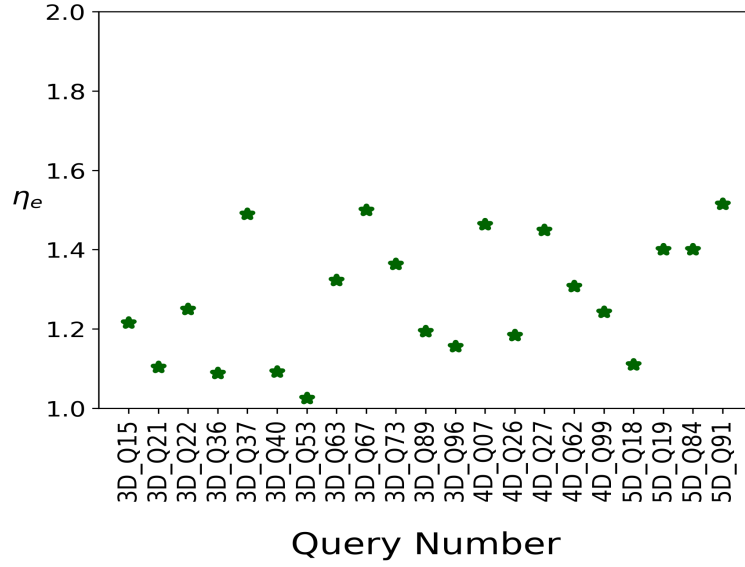Figure 7.7: Theoretical and Empirical Overheads Reduction ($\eta = 2$)

Figure 7.8: Empirical MSO Ratio ($\eta = 2$)

7.6.4 Validation of MSO Relaxation Constraint

A legitimate concern about FSB could be that while it guarantees maintenance of the η constraint in the theoretical framework, the MSO relaxation may exceed η in the *empirical* evaluation. To assess this possibility, we explicitly evaluated the empirical MSO ratio, η_e, incurred by FSB relative to SB. This was accomplished by exhaustively considering each and every location in the ESS to be q_a, and then evaluating the sub-optimalities incurred for these locations by SB and FSB. Finally, the maximum of these values was taken to represent the empirical MSO of each algorithm.

Contrary to our fears, the η_e values of FSB are always within $\eta = 2$, as shown in Figure 7.8. In fact, the η_e factors are within 1.5 for all queries. The main reason for the low η_e values in practice is due to the aggressive half-space pruning at each contour, and especially so at the final contour.

7.6.5 Dependency of γ on η

Thus far, we have analyzed the FSB results for the specific η setting of 2. We now move on to evaluating the γ behavior for different settings of η. This tradeoff is captured in Figure 7.9 for η values ranging over $[1, 3]$ for three different queries – Q15, Q27 and Q19 – with ESS dimensionalities of 3, 4 and 5, respectively.

We see an initial exponential increase in overheads reduction while going from $\eta = 1$ to $\eta = 2$, but this increase subsequently tapers off for larger values of η. For 3D_Q15, the number of optimization calls decreases steeply from 10^6 to 7010 when η is increased from 1 to 2, and then goes down marginally to 2950 calls when η is further increased to 3. The plateauing of the improvement with increasing η is because a certain minimum number of optimization calls is required for the basic functioning of the FSB algorithm.

7.6.6 Wall-Clock Time Experiments

All the experiments thus far assessed the $\gamma - \eta$ profile in the abstract world of optimizer cost values. We now present an actual execution experiment, where the end-to-end real-time performance (i.e. wall-clock times) was explicitly measured for the FSB and SB algorithms. Our representative example is based on TPC-DS Q19 featuring 5 error-prone predicates.

As mentioned previously, the task of identifying the contours is inherently amenable to parallelism. Even after exploiting this feature on a 64-core workstation platform, SB took a *few days* to identify all the contours for 5D_Q19. In marked contrast, a parallel version of the BCS identification in FSB, which utilizes the fact that there are $(D * \log_\eta(\frac{1}{\epsilon}))^{D-2}$ independently-explorable $2D$ segments per contour, completed the identification within *10 minutes* (for $\eta = 2$).

After building the ESS, it took SB around 20 mins to complete its query execution, incurring a sub-optimality of 4.8. On the other hand, FSB completed in around 26 mins, resulting in a sub-
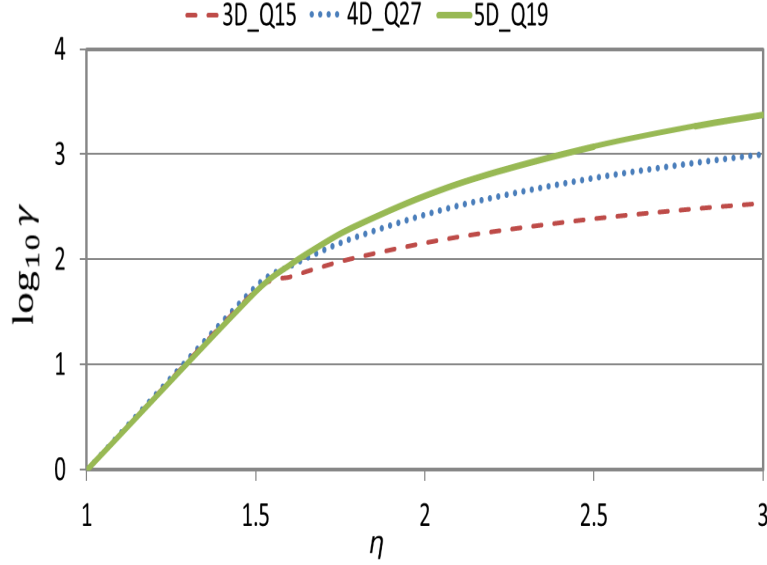
Figure 7.9: `FSB` Tradeoff (Theoretical)

optimality of 6.2. The drilled-down information of plan executions for every contour with `FSB` can be seen in Table 7.2.

So, overall, SB took days to create the `ESS` and execute this instance of Q19, whereas `FSB` required only $(10 \text{ minutes } + 26 \text{ minutes }) = 36 \text{ minutes }$ to complete the entire query processing. This means that even if the ad-hoc query eventually turns out to be a canned query, it would take more than 500 successive invocations before SB begins to outperform `FSB`.

We conducted additional experiments to establish the practicality of the `FSB` approach. Specifically, on a representative set of queries, we profiled `FSB` for its memory usage, CPU usage, and end-to-end latency. The memory usage is also a function of the server's database configuration, which was set with the PostgreSQL tuning tool [PGT]. The results, presented in Table 7.4, demonstrate that `FSB`'s resource requirements are reasonable and easily justified by the substantive performance benefits that it delivers. Moreover, the CPU usage is relatively small compared to the end-to-end latency since our database environment is disk-bound.

7.6.7 JOB Benchmark Results

The results on JOB benchmark are on similar lines to that of TPC-DS benchmark – one of the primary reasons being that the theoretical reduction in overheads are just a function of (D, r) for a fixed η. Table 7.3 captures the overheads reduction achieved with the same algorithmic parameters (such as r, and $\eta = 2$) used for TPC-DS.

The results again show that the theoretical reduction in overheads is least two orders of magnitude,

Table 7.2: FSB EXECUTION ON TPC-DS QUERY 19

Contour no.	e_1	e_2	e_3	e_4	e_5	Time (sec.)
1	-	p_1	-	p_2	p_3 **(100)**	54.1
2	-	p_4	-	p_5	-	122.5
3	-	p_4 **(1)**	-	-	-	182.1
3	-	-	-	p_5	-	251.4
4	-	-	-	p_5	-	357.8
5	-	-	-	p_5	-	509.9
6	p_6 **(5)**	-	-	-	-	789.2
6	-	-	p_7 **(96)**	-	-	1051.6
7	-	-	-	p_8 **(99)**	-	1562

Table 7.3: RESULTS ON JOB BENCHMARK WRT γ

Query	Theoretical	Empirical
3D_Q1a	179.8	826.4
4D_Q13a	332.9	2444.7
4D_Q23c	332.9	1972.8

and empirical reductions are around three orders of magnitude (especially, for the 4D queries).

Table 7.4: RESOURCE USAGE (100 GB)

Query	Memory Usage (MB)	CPU Times (mins)	Latency (mins)
3D_Q15	360	1.4	28.1
3D_Q96	220	1.3	17.8
4D_Q7	489	1.2	23
4D_Q26	490	1.5	12.6
4D_Q27	464	1.8	30.5
5D_Q19	1000	11	36
5D_Q84	348	2.8	10.1
5D_Q91	828	1.3	4.3

7.7 Related Work

In the prior robust query processing literature, there have been two strands of work – the first delivering savings on optimization overheads, and the other addressing the query execution performance. Given this context, FSB appears a unique proposition since it offers an attractive *tradeoff* between these two competing and complementary aspects. Since the related work wrt query execution is extensively covered in Chapter 2, we focus on compilation overheads below.

7.7.1 Compilation Overheads

As mentioned before, we measure the query compilation overheads in terms of the number of optimization calls made to the underlying database engine. With regard to this metric, the overheads incurred by SB in constructing the ESS can be computed as follows: SB first computes the optimal plans for *all* locations in the discretized ESS grid. This is carried out through repeated invocations of the optimizer with different selectivity values and combinations. Then, the isocost contours are drawn as connected curves on this discretized diagram. So, if we assume a grid resolution of r in each dimension of the ESS, the total number of optimization calls required by this approach is r^D.

Note, however, that we do not require the complete characterization of the ESS, but only the parts related to the isocost contours. An optimized variant, called **Nexus**, was proposed in [DH16] to implement this observation, and shown to make material reductions in the contour identification overheads. However, we have not included Nexus in our current study for the following reasons: (1) When a large number of contours are present in the ESS, which can happen if the cost at the

`ESS` terminus is much larger than at the origin, the net effort by Nexus in contour identification effectively becomes close to complete enumeration. (2) If a lower bound on a query's location in the `ESS` happens to be known through domain knowledge, SB can take advantage by making the lower bound to be the origin and thereby shrinking the `ESS`. However, the isocost contours would have to be redrawn from scratch by Nexus. (3) To provide performance fairness to queries across the `ESS`, *randomized* placement of contours was presented as a solution in [DH16]. In such cases, multiple sets of contours would have to be identified by Nexus, and it may cumulatively turn out to be more expensive as compared to complete enumeration. We have therefore chosen to instead simply assume that the entire `ESS` is enumerated, and consequently r^D is used as the baseline SB overheads in the sequel. Further, note that `FSB` is *not* impacted by such deployment issues since its compile-time efforts are carried out afresh at each ad-hoc query's submission time.

Other line of work wrt compilation overheads has been in the context of Parametric Query Optimization (PQO), where the objective is to have *precomputed* the appropriate plans for freshly submitted queries. In [HS03], the selectivity space was decomposed into polytopes that approximate plan-optimality regions, based on the geometric heuristic that "If all vertices of a polytope have the same optimal plan, then the plan is also optimal *within* the entire polytope". However, this assumption, as well as the presence of regular boundaries for the optimality regions, were later shown in [DDH08] to be largely violated in industrial-strength settings.

Instead of trying to characterize the entire selectivity space in advance, an alternative "pay as you go" approach was taken in [BBD09]. Here, the PQO overheads were restricted only on the actual query workload submitted to the system, facilitating a progressive and efficient exploration of the parameter space. In our setting, however, since we are apriori unaware of the query location, the BCS has to be constructed in an agnostic manner to this location.

More recently, a geometric property called Bounded Cost Growth (BCG) was identified in [DNC17], which typically holds on plan cost functions. In BCG, the relative increase of plan costs is modeled as a low-order polynomial function of the relative increase in plan selectivities. In fact, using the identity function for this polynomial is itself found to be generally satisfactory. Our use of concavity is similar to BCG in that, when the polynomial is the identity function, it is shown below that any PCF that satisfies APC also satisfies BCG.

7.7.1.1 BCG

Let us now see BCG's definition formally. For any PCF $\mathcal{F}_p$:

$$\mathcal{F}_p(\alpha * q.j) \leq f(\alpha) * \mathcal{F}_p(q.j) \quad \forall j \in \{1, \ldots, D\}, \forall \alpha \geq 1 \tag{7.4}$$

where $f(\alpha)$ is an increasing function and $\mathcal{F}_p(q)$ represents the cost of the corresponding plan at location q. In words, for any plan, if the selectivity is increased in any one of the dimensions by a factor $\alpha \geq 1$, then the cost of the plan also increases by a factor at most $f(\alpha)$. Moreover, they also claim that $f(\alpha) = \alpha$ would suffice in practice. As in the case of our axis-parallel concave assumption, they also show that if BCG holds true for POSP plans then it is also true for OCS.

7.7.1.2 Concavity implies BCG

Let us consider a PCF $\mathcal{F}_p$, if $\mathcal{F}_p$ is axis-parallel concave, then we will show that the PCF also satisfies the BCG assumption when $f(\alpha) = \alpha$. For this, we just need to show the implication over an 1D projection, which then easily generalizes for the generic scenario. Consider a location q, whose projection on dimension j (i.e., $q.j$) has slope m on $\mathcal{F}_p$. Thus the tangent at $q.j$ can be expressed as a line of the form $\mathcal{F}_p(q.j) = m * q.j + c$. However, $c \geq 0$ for the function value to be non-negative for $q.j = 0$. Hence, $\mathcal{F}_p(\alpha * q.j) \leq m(\alpha * q.j) + c \leq \alpha * (mq.j + c) = \alpha * \mathcal{F}_p(q.j)$,

7.8 Conclusions

Even though `DimRed` provides substantial reduction in compilation overheads, a major limitation of `SpillBound` is that the reduced overheads are still manageable for canned queries but remain too high for ad-hoc queries.

In this chapter, we address the above limitation by designing `FrugalSpillBound` whose compilation overheads are exponentially lower than those of `SpillBound`. Our construction of `FrugalSpillBound` is based on two basic principles: (a) leveraging the axis-parallel concave behavior exhibited by the PCFs and OCS with respect to predicate selectivities in the `ESS`; (b) substituting the original contours with much smaller contour-covering sets. Our theoretical analysis establishes a $\mu - \gamma$ tradeoff that is extremely attractive, delivering exponential improvements in γ for linear relaxations in μ. Further, the empirical improvements are even higher, by more than an order of magnitude. So, `FrugalSpillBound` takes an important step towards extending the benefits of MSO guarantees to ad-hoc queries.

Chapter 8

Deployment Aspects

In this chapter, we shall discuss some pragmatic aspects wrt usage of our proposed robust techniques in a real-world context, and also describe the complete workflow of our proposed robust database engine. For the deployment of the database engine, the following essential functionalities need to supported by it: (1) selectivity monitoring; (2) selectivity injection; (3) abstract plan costing and execution; (4) cost-budgeted partial execution of plans; and (5) spilling of plan executions. After describing these essential features in detail, we also discuss key additional deployment aspects such as `OptAssist` and parallel compilation. As mentioned before, our proposed algorithms are intrusive to the engine since they require changes to the core engine to support operator spilling and monitoring of selectivities. Our experience with PostgreSQL is that these facilities can be incorporated relatively easily – the full implementation required only a few hundred lines of code.

8.1 Essential Engine Features

8.1.1 Selectivity Monitoring

In every intermediate plan execution of our approach, we learn the lower bound on selectivity of an error-prone predicate (or it's actual value) through monitoring of selectivities. In PostgreSQL, we achieve this using the plan's `tuple count` data structure. Once the plan execution terminates (or, finishes), the monitored selectivity value contains the lower bound on selectivity of an `epp` (or, the `epp`'s actual selectivity value).

8.1.2 Selectivity Injection

For identification of isocost contours or bounded contour-covering sets, we need to be able to inject the desired selectivities for query predicates instead of the optimizer estimated values. One non-intrusive option is to suitably modify, for each new location in the `ESS`, the query constants and

the data distributions, but this is clearly 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 computation. Specifically, the optimizer estimated values are replaced with the injected ones where the estimates are computed for later use. Interestingly, some commercial optimizer APIs already support such selectivity injections to a limited extent (e.g. IBM DB2 [DB2]).

8.1.3 Abstract Plan Costing and Execution

In Chapter 7, we saw that abstract plan costing feature is required to compute the slope of plan cost functions at `ESS` locations. Further, during the execution phase, we need to be able to instruct the execution engine to execute a particular plan. The abstract plan costing feature is implemented by pruning away all but one plan in the engine's search space. This plan is essentially the abstract plan to be costed and executed. To prune away the search space the GUC (Grand Unified Configuration) parameters of PostgreSQL are used. The above features are currently provided by a few commercial systems (e.g., Microsoft SQL Server [Ser]).

8.1.4 Cost-budgeted Executions

As mentioned in earlier chapters, all the intermediate plan executions in our approach terminate if they exceed their assigned cost budget. One way is to have a *cost-instrument* which monitors the progressive cost during plan execution. The other way is to use a *timer* that keeps track of the time elapsed. Although bouquet identification provide budget in terms of abstract optimizer cost units, they can be converted to equivalent time budgets through the techniques proposed in [WCHN13]. No material changes need to be made in the engine internals in the timer scenario. The premature termination of plans can be achieved easily using the statement.cancel() functionality supported by JDBC drivers.

8.1.5 Spilling

As described in Section 4.2, spilling is achieved by deliberately breaking the operator pipeline at a specific node in the plan tree. This ensures that the downstream nodes do not get any data/tuples to process in order to achieve half-space pruning in the `ESS`. Equivalently the pipeline can be broken by just executing the sub-plan rooted at a specified node (leveraging the demand driven iterator execution model in place in PostgreSQL). Note that in each of our proposed algorithm, the final plan execution is not a spilled version, and hence after its execution the results are returned to the user.

In addition, `AlignedBound` requires a feature to get a least cost plan from optimizer which spills on a user-specified `epp`. This is implemented by pruning sub-plans in the DP lattice that violate

the rule of having another `epp` in the sub-tree rooted at the specified `epp`.

8.2 Efficiency Features

8.2.1 Parallelizing Compilation Phase

Construction of the contours in the `ESS` can be speed-ed up by leveraging the multiplicity of hardware to generating them in parallel since each call to the optimizer is independent of each other. Similar enhancement, is also applicable for finding bounded contour-covering sets.

8.2.2 OptAssist

Despite the fact that our robust algorithms provides worst-case run-time bounds, typically the sub-optimality incurred for query instances may reach upto 10 (due to the multiple partial executions involved in the process). Given this, a crucial deployment issue is that, there exist real query work-loads where the performance of the native optimizer's chosen plan for a query instance happens to be close to its optimal plan. For instance, if the native optimizer's estimates turn out to be correct (or, *very close* to the actual ones), then its plan choice is certainly a better alternative than any of our robust algorithms. So, we would prefer to execute the query using our robust algorithms only when the native optimizer is likely to have worse performance than the robust algorithms. Thus, for an input query instance, let us now see how to design `OptAssist` that help users to choose the better of native optimizer and our robust alternatives to process the query.

Approach

For ease of presentation, we choose `SpillBound` as a representative for the robust algorithms. `OptAssist` is not affected by this choice since any of our robust algorithms provide MSO guarantees just by query inspection. We approach this problem by first identifying the set of effective error-prone predicates (`epps`) for the input query using the `DimRed` component, specifically, until `MaxSelRemoval` of the `DimRed` pipeline. The next step is to *predict the risk* of the native optimizer chosen plan P, denoted by $MSO_{plan}(P)$ metric. Along with this, we also propose a relaxed metric for plan risk, $MSO^{80}_{plan}(P)$, which captures the worst sub-optimal performance in most (say, 80% of the cases – assuming uniform distribution of error scenarios) rather than all cases. Using these above risk values along with `SpillBound`'s MSO guarantee, i.e., $MSO_{SB} = D^2 + 3D$, the user can then make a choice of the query path, accordingly. The good news is that all the above mentioned values, $MSO_{plan}(P)$, $MSO^{80}_{plan}(P)$ and MSO_{SB} can be computed efficiently, thus making it viable for ad-hoc queries.

Problem Framework

We use $MSO_{plan}(P)$ and $MSO^{80}_{plan}(P)$ metrics to assess the risk of a plan P, which are formally defined below:

$$MSO_{plan}(P) = \max_{q_a \in \texttt{ESS}} \frac{Cost(P, q_a)}{\text{COST}(q_a)} \tag{8.1}$$

In essence, $MSO_{plan}(P)$ captures the worst-case impact on sub-optimality due to estimation errors of using plan P for the input query. In other words, the metric says that the sub-optimality incurred by using plan P, for the actual selectivity q_a being anywhere in the $\texttt{ESS}$, is less than or equal to $MSO_{plan}(P)$ value. Here, we assume that estimation errors, i.e., q_a can lie anywhere in the $\texttt{ESS}$ with equal probability. This assumption is justified from the fact that the cardinality model in real system can easily induce orders of magnitude estimation errors [Loh14, LGM+15]. Furthermore, approaches such as [Hue] and [WBM+18] also make uniformity assumption on the distribution of q_a in the $\texttt{ESS}$ to assess plan riskiness wrt execution performance.

At this juncture, a natural OptAssist is to check the condition: $MSO_{plan}(P) < MSO_{SB}$. If this is true, then choose P to execute; else, choose SpillBound for query execution. An interesting observation is that, it is often the case that for most q_a's in the $\texttt{ESS}$ the corresponding sub-optimality is less than MSO_{SB} but for few locations it becomes much more than MSO_{SB}. This is captured, in Figure 8.1, by the cumulative distribution function (CDF) of maximum sub-optimality wrt the percentage of the $\texttt{ESS}$ locations covered by a native optimizer chosen plan from TPC-DS Q91 query instance. Here, if the user wants to cover 80% of the $\texttt{ESS}$ locations then the maximum sub-optimality is 4.2. However, the value shoots up to 707.6 if all the locations needs to be covered. To demonstrate that the above behaviour is not a isolated instance, we considered the CDF function over our entire suite of queries. The results show that, on an average, for around 3000 plans, the value increases by a factor of around 100 when one wants to cover 100% locations from 80% locations.

Thus, to handle scenarios where the user is interested in covering most cases rather than all, we consider a relaxed version of $MSO_{plan}(P)$. This is captured by $MSO^{80}_{plan}(P)$ for any plan P which covers most, say 80%, of the $\texttt{ESS}$ locations. Thus,

$$MSO^{80}_{plan}(P) = 80^{th} \text{ percentile value in } \{\frac{Cost(P, q_a)}{\text{COST}(q_a)}, \forall \ q_a \in \texttt{ESS}\} \tag{8.2}$$

In this case, the modified condition for OptAssist could be $MSO^{80}_{plan}(P) < MSO_{SB}$. Given the above framework, our next objective is to efficiently compute $MSO_{plan}(P)$ and $MSO^{80}_{plan}(P)$, to support ad-hoc queries.
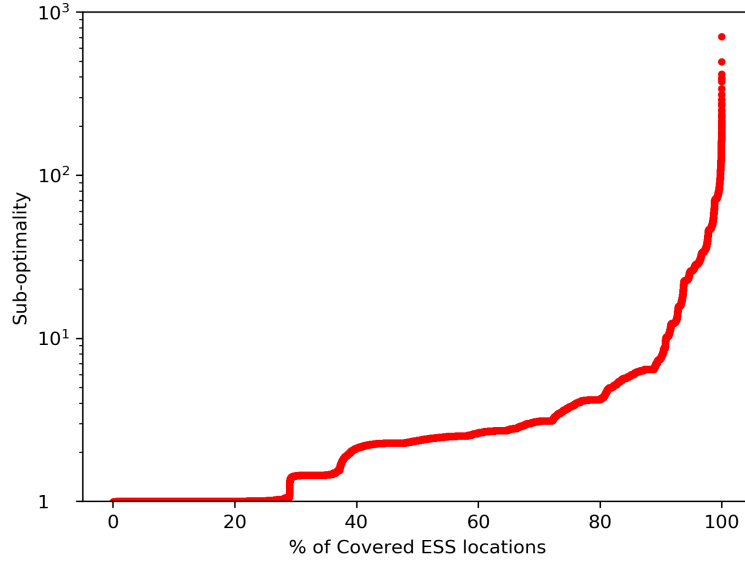
Figure 8.1: Cumulative distribution function of the sub-optimalities wrt ESS Coverage

Efficient Computation of Metrics for Plan Risk

Now we show that both the above mentioned plan risk metrics can be efficiently computed. Specifically, we theoretically prove that $MSO_{plan}(P)$ is located at one of the corners of the ESS. Further, $MSO_{plan}^{80}(P)$ can be computed by making optimizer calls at a small *perimeter band* of the ESS.

$MSO_{plan}(P)$: To efficiently compute $MSO_{plan}(P)$, the key idea is that, for any axis parallel 1D line segment in the ESS, the maximum value of the sub-optimality values occurs at one of the two end points. This follows by assuming that OCS and Plan Cost Functions (PCFs) satisfy axis-parallel piece-wise linearity assumption and certain slope behaviour among these pieces. Let us now prove this formally.

Theorem 8.1 $MSO_{plan}(P)$ *for any plan P occurs at one of the vertices of the* ESS.

Proof: The proof for this theorem is along the same lines as that of Theorem 6.1. For simplicity, we consider the 2D scenario with dimensions X and Y, and present the proof in brief, highlighting the major differences from the earlier proof.

We use the sub-optimality function, f_s, for a plan to denote the sub-optimality of the plan at an ESS location. Formally, the sub-optimality function, $f_s(P, q)$ of plan P, at location $q \in$ ESS, is defined as:

$$f_s(P, q) = \frac{Cost(P, q)}{\text{Cost}(q)}$$

130

Thus, we can define $MSO_{plan}(P)$ in terms of the sub-optimality function as:

$$MSO_{plan}(P) = \max_{q_a \in \text{ESS}} \{f_s(P, q)\} \tag{8.3}$$

Now, our objective is to show that the maximum value of the sub-optimality function, f_s, occurs at the vertices of an ESS. In order to achieve this, we do the following: (a) Prove the above with axis-parallel linear assumption (APL) of OCS and PCFs; (b) However, the linearity is often not true in practice. Thus, we extend the previous result by relaxing the linearity assumption to its piece-wise equivalent, and assuming that these linear pieces follow a certain empirically validated slope behaviour.

Axis-parallel Linear Assumption   Let us now focus on the case when the OCS and the PCFs follow APL property. We use the following lemma that, for any axis-parallel linear segment, the maximum value of the sub-optimality function f_s occurs at the end points of the line segment. For ease of presentation, the lemma is visually captured in Figure 8.2.

Lemma 8.1 *Given a line segment $Y = q.y_c$ that is parallel to the X axis, the maximum value of f_s, among the points in the line segment, occurs at one of its end points: either $(0, q.y_c)$ or $(1, q.y_c)$.*

Proof: We request the reader to refer to the proof of Lemma 6.2 since the proof is on very similar lines. □
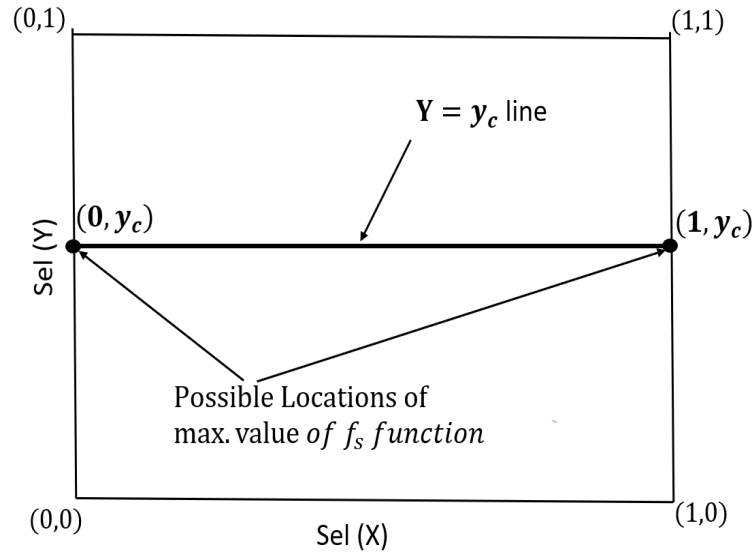

Figure 8.2: Maximum value of sub-optimality function captured at line segment's end points

Extending to the entire ESS from an individual segment, we get

131

Lemma 8.2 *Computing f_s along the vertices of an* `ESS` *is sufficient to establish f_s within the entire* `ESS`.

Proof: From the previous lemma, we know that for any axis-parallel lines segment, the local f_s occurs at one of its end-points. Let the lemma hold for a d-dimensional space. By mathematical induction we next show that the lemma also holds for a $d + 1$-dimensional space. Note that a $d + 1$-dimensional space can be viewed as two d-dimensional spaces where the corresponding vertices are connected by 1D lines. By the induction hypothesis the the local f_s computation can be moved to the vertices of each of the two d-dimensional spaces. Hence the lemma also holds for a $d+1$-dimensional space. $\qquad\qquad\square$

Relaxing Axis-parallel Linear Assumption As mentioned before, the APL assumption for OCS and PCFs does not always hold in practice. We relax the APL assumption to its axis-parallel piece-wise linear version (since any 1D function can be *approximately* divided into piece-wise linear segments). In Section 6.5.4, we empirically saw that OCS fitting with axis-parallel piece-wise linear property with less error. Hence we expect better fits with PCFs since it is a lesser complex function than OCS (OCS function is essentially the point-wise minimum of all PCFs).

With the piece-wise APL property, let us say that there are at most k pieces in any 1D segment of the OCS and PCF's in the `ESS`. Then, for each of the $2k$ pieces conditioned on PCFs and OCS, we make an assumption that for these $2k$ pieces the slope of the sub-optimality function is monotonically increasing or deceasing. This assumption is assessed by considering 1D segments over thousand of plans and corresponding OCS (chosen from our suite of queries), and evaluating if the pairwise slope-intercept product is consistently dominating across all the above $2k$ pieces (specifically, checking whether $cq \geq dp$ or $cq \leq dp$, corresponding to the numerator in Equation 6.9, is consistently true). The assessment shows that it is indeed true by more than 80% of the 1D segments in `ESS`, along all such $2k$ pieces. Thus we can claim that the f_s would still lie at one of the `ESS` corners for practical setting. $\qquad\qquad\square$

MSO$_{\textbf{plan}}^{\textbf{80}}$(P): Let us now turn our attention to the computation of $MSO_{plan}^{80}(P)$ metric. Instead of the vertices, we next show empirically that we need to consider a small *perimeter band* in an `ESS`. Here, perimeter band refers to the set locations in the `ESS` grid which are at a small axis-parallel distance from any perimeter point of the `ESS`. Specifically, if we say that the band is size b, then it is all the `ESS` locations which are at a axis-parallel distance b from any point of the perimeter. The perimeter and perimeter band locations for an example 2D `ESS` (with epps X and Y) are illustrated in Figure 8.3. The green coloured 1D segments correspond to the perimeter, whereas, additionally, blue colored segments correspond to the perimeter band for $b = 2$. After computing the set of sub-optimalities along the perimeter band, and from this a carefully chosen value is set as $MSO_{plan}^{80}(P)$,

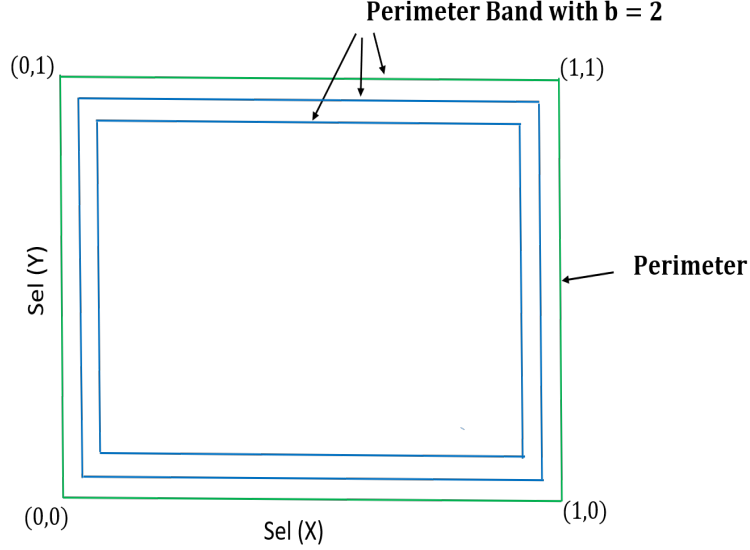as explained later in the Subsection Empirical Evaluation.

Figure 8.3: Perimeter Band of the `ESS`

Now that we know an efficient way of computing the plan riskiness metrics, the user could use the following `OptAssist` algorithm, whose pseudocode can be seen in Algorithm 6. Since, $MSO_{plan}(P)$ covers all the `ESS` locations and easier to compute than $MSO_{plan}^{80}(P)$, first check the $MSO_{plan}(P) < MSO_{SB}$ condition. If this is true, then choose the native optimizer for query execution; if not, then go on to the check second condition $MSO_{plan}^{80}(P) < MSO_{SB}$. If this is true, again choose native optimizer; else, choose `SpillBound` for query execution.

Empirical Evaluation

We empirically evaluate the accuracy of our proposed techniques for estimating $MSO_{plan}(P)$ and $MSO_{plan}^{80}(P)$ values. We have used the Q-Error [MNS09] as our error metric, wherein the error value for two real values a, b is the following:

$$Q\text{-}Error(a, b) = Max(a/b, b/a)$$

The assessment is performed on over 3000 POSP plans generated from 21 TPC-DS query templates.

Accuracy of Plan Riskiness Metrics: Figure 8.4 captures the average Q-error of the predicted $MSO_{plan}(P)$ using the `ESS` corners and the actual $MSO_{plan}(P)$ value enumerated using the entire `ESS`, over all the considered plans P in our suite.

We see that the Q-error is equal to 1 for most queries with minor violations in few. The results

133

Algorithm 6 `OptAssist` Algorithm
___
1: *Input:* Query Instance q;
2: Obtain the native optimizer P for q;
3: Compute $MSO_{plan}(P)$ using the corners of the `ESS`;
4: **if** $MSO_{plan}(P) < MSO_{SB}$ **then**
5:     Choose "Native Optimizer" for q's execution;
6:     **Return**;
7: **end if**
8: Compute $MSO_{plan}^{80}(P)$ using the perimeter band of width b of the `ESS`;
9: **if** $MSO_{plan}^{80}(P) < MSO_{SB}$ **then**
10:     Choose "Native Optimizer" for q's execution;
11: **else**
12:     Choose "`SpillBound`" for q's execution;
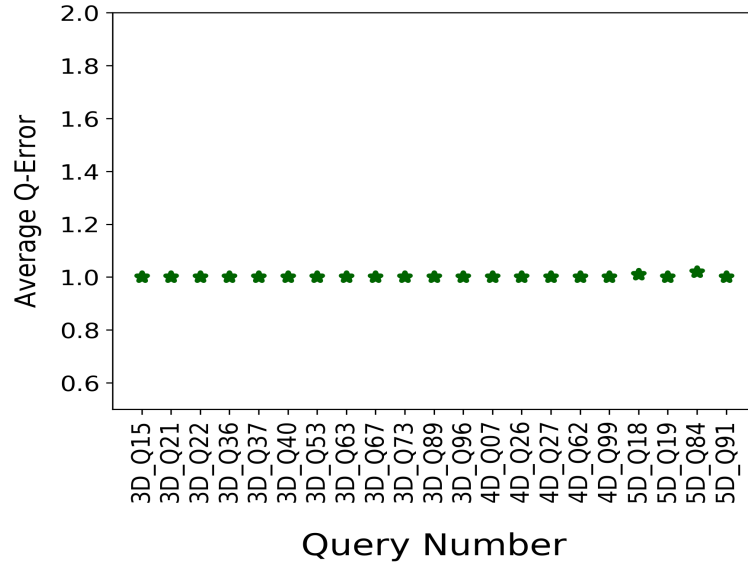13:   **Return**;
14: **end if**
___

Figure 8.4: Accuracy of $MSO_{plan}(P)$ value predicted from `ESS` Corners

suggest that, also confirming our theoretical analysis, using the vertices of the ESS, we can almost accurately predict the actual $MSO_{plan}(P)$ value. The minor errors, for instance with query template 5D_84, is because some plans violate the fundamental PCM property.

Moving to the relaxed version of the above metric, i.e., $MSO_{plan}^{80}(P)$, we begin with enumerating the sub-optimalities using the perimeter band of an ESS. We choose the value corresponding to 2/3rd-percentile (an empirically chosen value) from the set of sub-optimalities, and set it to $MSO_{plan}^{80}(P)$. The accuracy of the predicted $MSO_{plan}^{80}(P)$ value, captured in terms of the Q-error, is shown in the Figure 8.5. We observe that we were able to successfully predict the $MSO_{plan}^{80}(P)$ value with an average Q-error of less than 1.5 for most queries. Further, we see that a value of $b = 4$ for resolution of 100 suffices for all queries, and the time taken to compute the band for any query is few minutes using parallelization on a 64-core machine.
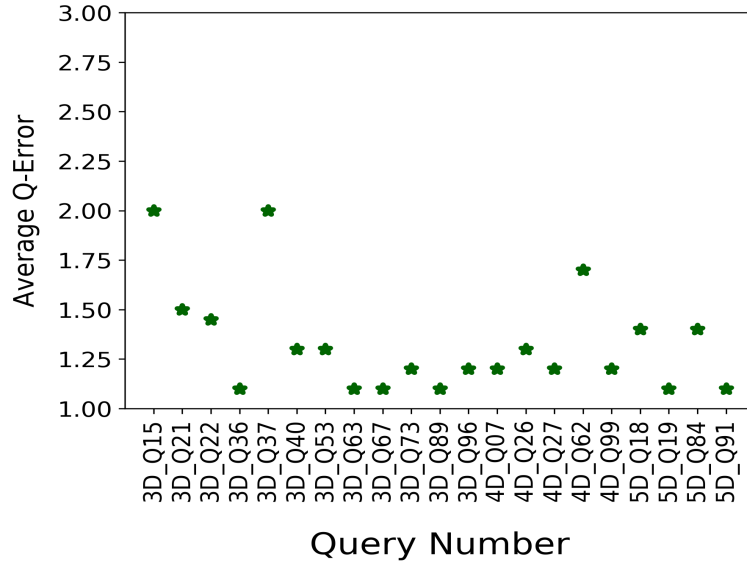

Figure 8.5: Accuracy of $MSO_{plan}^{80}(P)$ value predicted from ESS perimeter band

8.3 Relaxing Perfect Cost Model Assumption

Finally, another deployment issue is that until now we have assumed cost model to be perfect, but this assumption is hardly true in practice. However, if we were to be assured that the cost modeling errors, while non-zero, are *bounded* within a δ multiplicative error factor, then the MSO guarantees in this thesis will carry through modulo an inflation by a factor of $(1+\delta)^2$. For example, the MSO guarantee of SpillBound would be $(D^2 + 3D)(1+\delta)^2$. Moreover, the errors induced by cost model are fairly small, for instance, $\delta = 0.4$ is reported in [WCHN13], resulting in doubling of MSO.
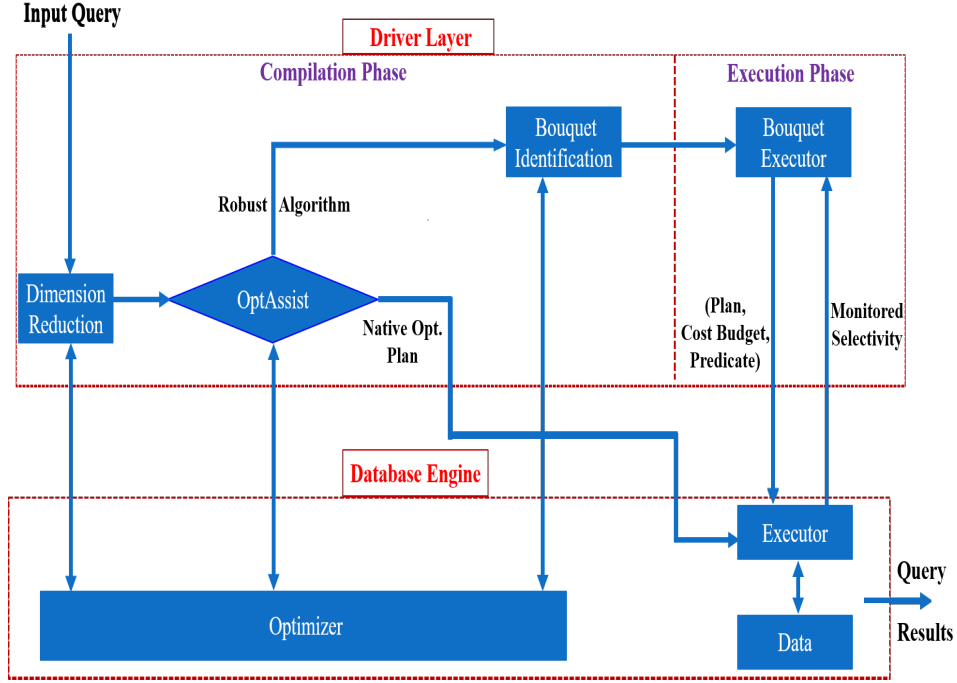
Figure 8.6: Architecture of Proposed Robust Database Engine

8.4   Architecture Description

The flow diagram of the architecture of our proposed robust database engine is captured in Figure 8.6. The query first goes through the `DimRed` component to identify the `epps`. Then it is fed into the `OptAssist` component which aids the user to choose either the native optimizer plan or our robust algorithm for its query execution. If the native optimizer is chosen by the user then the query is executed with the corresponding plan and the query results are returned. In the other case, either `SpillBound` or `FrugalSpillBound` is chosen based on the query being canned or ad-hoc, respectively.

Once a robust algorithm is chosen, then it is followed by the construction of contours or bounded contour-covering sets by interacting with the query optimizer as part of its compilation phase. Then, in the execution phase, the chosen algorithm selects the (*plan*, *budget*, *predicate*) for every execution. The triplet information corresponds to the *plan* being executed with the cost *budget* spilling on the *predicate*. The first plan is execution based on the triplet information corresponding to the origin of the `ESS`, i.e., assuming zero selectivity for all the `epps`. Based on the selectivity information learnt from this execution (either the lower bound or actual selectivity of the `epp`) the next plan is chosen. This sequence is repeated until a plan reaches completion returning the query results to the user.

8.5 Performance Comparison b/w Native Optimizer and Proposed Robust Techniques

To asses the benefits of our proposed techniques delivered at run-time, we carried out experiments wherein query response times of `SpillBound` (an illustrative robust algorithm) were explicitly measured in comparison to the native optimizer (PostgreSQL). We use the example query (EQ), as shown in Figure 1.1(a), and Q32 and Q19 as the representative set of queries from the TPC-DS benchmark. Here, the number of joins in the queries go up to 5. The results of the evaluation, captured in Figure 8.7, show that the execution time of `SpillBound` is consistently less than 50% as compared to PostgreSQL, the running time of which is normalized to 1.
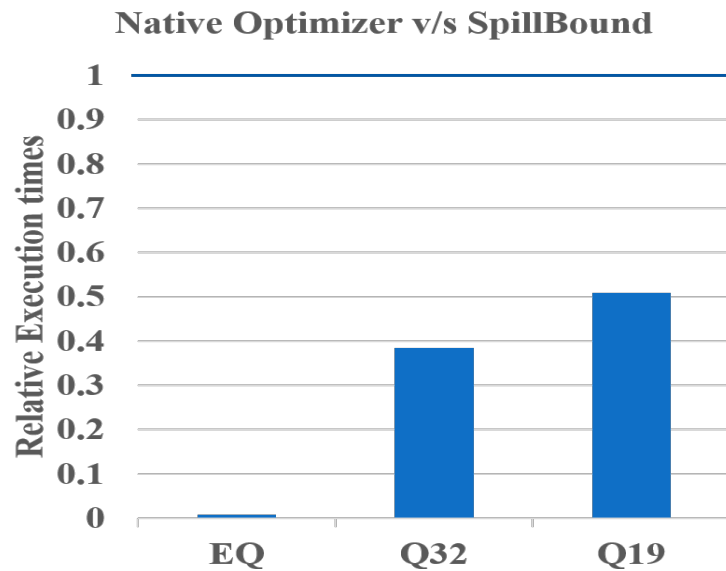

Figure 8.7: Comparison of execution times

Chapter 9

Conclusions and Future Work

It is a folklore in database research that errors in selectivity estimates result in poor choice of query execution plans, leading to orders of magnitude slowdown in query performance. To address the classical selectivity estimation problem, a different approach called `PlanBouquet` was proposed in 2014, wherein the estimation process is completely abandoned and replaced with a calibrated discovery mechanism. The beneficial outcome of this new construction is that it lends itself to obtaining provable guarantees on worst-case query execution performance.

However, `PlanBouquet` has several key limitations on compilation and execution fronts for practical use. These limitations include: (a) huge compilation efforts to be amenable for MSO guarantees, (b) no support for high-dimensional queries, and (c) variable MSO guarantees across platforms. The primary contribution in this thesis is a set of robust query processing algorithms that: (i) provide strong and platform-independent MSO guarantees, (ii) handle high-dimensional queries, and (iii) support ad-hoc queries with low compilation overheads. Further, we present a `OptAssist` heuristic to aid the user in choosing between the native optimizer and our proposed robust algorithms.

We next present our conclusions in more detail. Later on we describe some directions for further research to achieve robustness in data management.

9.1 Conclusions

In the first segment of the thesis, we presented `SpillBound`, a query processing algorithm that delivers an MSO guarantee of $D^2 + 3D$, which is dependent solely on the dimensionality of the selectivity space. This substantive improvement over `PlanBouquet` is achieved through a potent pair of conceptual enhancements: half-space pruning of the ESS thanks to a spill-based execution model, and bounded number of executions for jumping from one contour to the next. Our new approach facilitates porting of the bound across database platforms, and that the bound is easy to compute as

138

we could merely do it by query inspection. Further, it has low magnitude and is not reliant on the anorexic reduction heuristic. We also showed that `SpillBound` is within an $\mathcal{O}(D)$ factor of the best deterministic selectivity discovery algorithm in its class. Finally, we introduced the contour alignment and predicate set alignment properties, and leveraged them to design `AlignedBound` with the objective of bridging the quadratic-to-linear MSO gap between `SpillBound` and the lower bound.

A detailed experimental evaluation on industrial strength benchmark queries demonstrated that our algorithms provide competitive guarantees to `PlanBouquet`, while their empirical performance is significantly superior. The performance of `AlignedBound` is even better than `SpillBound` while achieving single-digit MSO guarantees.

In the subsequent segment, we overcome an important limitation of the `SpillBound` class of techniques. Namely, these techniques are not operational for high-dimensional selectivity spaces since: (a) their compilation overheads are exponential in the dimensionality of the space, and (b) their performance bounds are quadratic in the dimensionality. We address this dimension limitation by presenting the `DimRed` pipeline. Our proposed method systematically reduces seemingly high-dimensional queries to low-dimension equivalents without sacrificing the performance guarantees.

The `DimRed` pipeline, which leverages schematic, geometric and piggybacking techniques to reduce even queries with more than 15 dimensions in the selectivity space to five or less dimensions. In fact, for quite a number of queries, the dimensionality came down to the lowest possible value of 1! Gratifyingly, not only could we dramatically decrease the overheads due to such reductions, but could also significantly improve the quality of the MSO guarantee.

The drastic reduction in overheads attained by `DimRed` made `SpillBound` practical for canned queries that are repeatedly invoked by the parent application. However, for ad-hoc queries which are issued on the fly, the overheads proved to be still too high. We addressed this limitation, in the subsequent chapter, by designing `FrugalSpillBound` that provides a trade-off between MSO guarantees and compilation overheads. Our theoretical analysis establishes that the trade-off is extremely attractive, delivering exponential improvements in compilation overheads for linear relaxations in MSO guarantees. This is attained by leveraging the concave-down property of plan cost functions over the `ESS`. The empirical improvements are even better, delivering a cumulative benefit of more than three orders of magnitude for just a doubling in the MSO guarantee.

Finally, for increasing the efficacy of our robust alternatives, we proposed `OptAssist` for assisting the user to choose between the native optimizer or our proposed robust alternatives. This becomes important since it is possible that real-world workloads may include query instances for which the MSO of the native optimizer plan is very low, in which case our proposed techniques would turn out to be a sub-optimal choice. Currently, our assist provide initial directions to address this problem.

To summarize, this thesis proposed query processing algorithms based on a potent set of geometric

search techniques to achieve theoretical and practical guarantees on query performance. Thereby, take a substantive step forward in making robust query processing a contemporary reality.

9.2   Future Work

We now move on to enumerating a set of future research directions, some of which are borrowed from [Har19] with the author's consent. Solutions to some of these issues would take significant steps further towards the ultimate quest for robust query processing. We would like to note that the future directions mentioned below are applicable to the `PlanBouquet` technique as well.

1. **Handling Database Updates:** Our proposed robust algorithms' compilation phase are inherently robust to changes in data distribution, since these changes only shift the location of q_a in the `ESS`. However, 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, the development of incremental and efficient techniques for bouquet maintenance is essential for catering to dynamic databases.

2. **Graceful Performance Degradation:** A major problem faced in real deployments is the presence of "performance cliffs", where the performance suddenly degrades precipitously although there has only been a minor change in the operational environment. This is particularly true with regard to hardware resources, such as memory. So, an important future challenge is to design algorithms that provably degrade gracefully with regard to all their performance related parameters.

3. **Handling Dependent Predicate Selectivities:** In this thesis, we have assumed independence in predicate selectivities. However, this assumption often does not hold in practice even though it is widely made in the literature. Therefore, extending our techniques to handle dependent selectivities with strong MSO guarantees would certainly be one of the important future directions.

4. **Robust Query Processing at Large Scale:** The goal of large scale query optimization is to reduce the optimization time of queries with large number of joins, handling in excess of 100 tables [NR18]. A natural extension is to see whether execution-time guarantees could be provided w.r.t. the *optimal* plan for such large queries, possibly leveraging some of our results.

5. **Extension to Non-relational Systems:** In recent times, the database community has seen a large number of attempts to suit different kinds of data processing requirements that do not resemble the traditional relational database systems. Further, it is well known that many crucial

optimizations, such as join order, are not well supported in these Data-Intensive Scalable Computing (DISC) systems because they lack the necessary data statistics [LLDI18]. We wish to highlight that the proposed techniques are useful to any system that: (i) uses a cost-based optimizer to choose the ideal execution plan as a function of selectivities, and (ii) uses a bottom-up and sequentially pipelined based executor to support spilling of plan executions.

As a case in point, in graph databases the query optimization problem gets more complex due to sparse statistics. Since, specialized graph database systems such as Neo4j support both cost-based optimization and bottom-up pipelined based execution [Neo], we hope that our proposed techniques can be ported to these systems to achieve robust performance. Moreover, on the other extreme end, there are relational systems which natively support graph analytics, in which case, our proposed techniques could be even easier to port. Further, the type of queries issued for graph databases are different than that of relational databases. However, there are some initial works which can convert the graph queries into SQL equivalents for Apache TinkerPop (a graph computing framework for both transactional and analytical graph database systems) [Tin]. Finally, these SQL versions of graph queries need to be along the lines of typical analytical processing benchmark (such as TPC-H and TPC-DS) queries for the porting. However, it will be interesting future work to extend these techniques for the larger domain of queries which are handled in graph database systems.

6. **Machine Learning for Robust Query Processing:** Recently, machine learning techniques have been applied to query processing and optimization in RDBMS. However, none of these ML techniques currently provide strong guarantees on worst-case query execution performance. Thus, it is an interesting research direction to provide such guarantees in the quest of robust query processing.

7. **Multi-dimensional Online Bidding Problem:** A classical problem in the theoretical computer science literature is the *online bidding problem* [CKNY08]. The problem is as follows: In the face of an unknown threshold $T \in R^+$, an algorithm must submit *bids* $B \in R^+$ until it submits a bid $b \geq T$. The goal is to minimize the competitive ratio, i.e. ratio of sum of bids paid over the unknown threshold. This problem was successfully mapped to the 1D version of `PlanBouquet` (or equivalently, `SpillBound`) to get a matching upper and lower bound of 4 on the MSO guarantee.

The multi-dimension version of online bidding problem can be viewed as each bid being a D-dimensional vector. Further, the termination criteria being that the final bid vector dominates the threshold in all the dimensions. Currently, we are not aware of any general results to this

problem.

A rich set of interesting theoretical problems arise from the original multidimensional online bidding problem by considering different class of *bidding functions* and *rules*. Note that `SpillBound` can also be mapped to this multi-dimension problem constrained to a particular set of bidding rules.

Bibliography

[AC99]   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*, pages 181–192, 1999. 6, 14, 16

[AÇR⁺12] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. Learning-based query performance modeling and prediction. In *Proc. of the 28th IEEE Intl. Conf. on Data Engg., ICDE '12*, pages 390–401, 2012. 20

[AH00]   Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, 2000. 18

[BBD05]   Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive re-optimization. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pages 107–118, 2005. 6, 14, 17

[BBD09]   Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive parametric query optimization. In *IEEE Trans. on Knowledge and Data Engineering (TKDE), 21(4)*, pages 582–594, 2009. 124

[BC05]   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*, pages 119–130, 2005. 6, 14

[BCG01]   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*, pages 211–222, 2001. 16

[BGIA⁺18] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth scan: robust access path selection without cardinality esti-

mation. In *Intl. Journal on Very Large Data Bases (VLDB), 27(4)*, pages 521–545, 2018. 19

[CGG04] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proc. of the 20th IEEE Intl. Conf. on Data Engg., ICDE '04*, pages 227–238, 2004. 75

[Cha09] Surajit Chaudhuri. Query optimizers: time to rethink the contract? In *Proc. of the 2009 ACM SIGMOD Conf. Intl. Conf. on Management of Data*, pages 961–968, 2009. 5

[CHG02] 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. 6, 14

[CKNY08] Marek Chrobak, Claire Kenyon, John Noga, and Neal E. Young. Incremental medians via online bidding. In *Algorithmica, 50(4)*, pages 455–478, 2008. 141

[CNR04] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 803–814, 2004. 34

[CY17] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *Proc. of the 2017 ACM SIGMOD Intl. Conf. on Management of Data*, pages 759–774, 2017. 16

[dag10] Dagstuhl Seminar. Robust Query Processing. www.dagstuhl.de/en/program/calendar/semhp/?semnr=10381, 2010. 5

[dag12] Dagstuhl Seminar. Robust Query Processing. www.dagstuhl.de/en/program/calendar/semhp/?semnr=12321, 2012. 5

[dag17] Dagstuhl Seminar. Robust Query Processing. www.dagstuhl.de/en/program/calendar/semhp/?semnr=17222, 2017. 5

[DB2] IBM DB2. Using a selectivity clause to influence the optimizer. www.ibm.com/developerworks/data/library/tips/dm-0312yip/. 127

[DDH07] Harish D., 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. 31, 46, 88

[DDH08] Harish D., Pooja N. Darera, and Jayant R. Haritsa. Identifying robust plans through plan diagram reduction. In *Proc. of the VLDB Endow., 1(1)*, pages 1124–1140, 2008. 6, 14, 124

[DH16] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: A fragrant approach to robust query processing. In *ACM Trans. on Database Systems (TODS), 41(2)*, pages 1–37, 2016. 6, 23, 24, 33, 45, 46, 73, 123, 124

[DNC17] Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. Leveraging re-costing for online optimization of parameterized queries with guarantees. In *Proc. of the 2017 ACM SIGMOD Intl. Conf.*, pages 1539–1554, 2017. 103, 124

[Dut] Anshuman Dutt. Plan bouquets: An exploratory approach to robust query processing. PhD Thesis, Indian Institute of Science, 2016, `https://dsl.cds.iisc.ac.in/publications/thesis/anshuman_thesis.pdf`. 14, 19

[DWN+19] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. In *Proc. of the VLDB Endow., 12(9)*, pages 1044–1057, 2019. 16

[GKD+09] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. of the 25th IEEE Intl. Conf. on Data Engg., ICDE '09*, pages 592–603, 2009. 19

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. In *ACM Computing Surveys, 25(2)*, pages 73–170, 1993. 34

[Gra12] Goetz Graefe. New algorithms for join and grouping operations. In *Computer Science - Research and Development, 27(1)*, pages 3–27, 2012. 19

[Har18] Jayant R. Haritsa. Robust query processing in database systems. In *Advanced Computing & Communications, 2(2)*, pages 13–21, 2018. 2

[Har19] Jayant R. Haritsa. Robust query processing: Mission possible (tutorial). In *Proc. of the 35th IEEE Intl. Conf. on Data Engg., ICDE '19*, pages 2072–2075, 2019. 5, 140

[HS02] Arvind Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases, VLDB '02*, pages 167–178, 2002. 22, 82

[HS03] Arvind Hulgeri and S. Sudarshan. Anipqo: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases, VLDB '03*, 2003. 124

[Hue] Fabian Hueske. Specification and optimization of analytical data flows. PhD Thesis, TU Berlin, 2016. 129

[IC91] 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*, pages 268–277, 1991. 3

[Ioa03] Yannis E. Ioannidis. The history of histograms (abridged). In *Proc. of the 29th Intl. Conf. on Very Large Data Bases, VLDB '03*, pages 19–30, 2003. 15

[KD98] 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*, pages 106–117, 1998. 6, 14

[KHBM17] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating join selectivities using bandwidth-optimized kernel density models. In *Proc. of the VLDB Endow., 10(13)*, pages 2085–2096, 2017. 16

[KKR+19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2019. 16

[KMSB15] Andranik Khachatryan, Emmanuel Müller, Christian Stier, and Klemens Böhm. Improving accuracy and robustness of self-tuning histograms by subspace clustering. In *IEEE Trans. on Knowledge and Data Engineering (TKDE), 27(9)*, pages 2377–2389, 2015. 16

[KYG+18] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. In *ArXiv preprint:1808.03196*, 2018. 16, 20

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? In *Proc. of the VLDB Endow., 9(3)*, pages 204–215, 2015. 3, 25, 51, 129

[LKNC12] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. In *Proc. of the VLDB Endow., 5(11)*, pages 1555–1566, 2012. 19

[LLDI18] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. Rios: Runtime integrated optimizer for spark. In *Proc. of the ACM Symposium on Cloud Computing, SoCC '18*, pages 275–287, 2018. 141

[LNS07] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases, VLDB '07*, pages 195–206, 2007. 75

[LNS09] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. Approximate substring selectivity estimation. In *Proc. of the 12th Intl. Conf. on Extending Database Technology, EDBT '09*, pages 827–838, 2009. 75

[Loh14] Guy Lohman. Is query optimization a solved problem? http://wp.sigmod.org/?p=1075, 2014. 3, 5, 6, 129

[LRG+17] Viktor Leis, Bernharde Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2017. 15, 17

[MGY15] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. Cliffguard: A principled framework for finding robust database designs. In *Proc. of the 2015 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1167–1182, 2015. 20

[MMK18] Magnus Müller, Guido Moerkotte, and Oliver Kolb. Improved selectivity estimation by combining knowledge from sampling and synopses. In *Proc. of the VLDB Endow., 11(9)*, pages 1016–1028, 2018. 15

[MNS09] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. In *Proc. of the VLDB Endow., 2(1)*, pages 982–993, 2009. 6, 14, 133

[MP19] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2019. 20

[MRS⁺04] 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*, pages 659–670, 2004. 3, 17

[Nau16] Jeffrey F. Naughton. Technical perspective: Broadening and deepening query optimization yet still making progress. In *ACM SIGMOD Record, 45(1)*, page 23, 2016. 5

[Neo] Neo4j. https://neo4j.com/blog/introducing-new-cypher-query-optimize 141

[NG13] Thomas Neumann and Cesar Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *Proc. of the 15th Conf. on Database Systems for Business, Technology, and Web, BTW '13*, pages 73–92, 2013. 6, 14

[NR18] Thomas Neumann and Bernhard Radke. Adaptive optimization of very large join queries. In *Proc. of the 2018 ACM SIGMOD Intl. Conf. on Management of Data*, pages 677–692, 2018. 20, 140

[OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proc. of the 2nd Workshop on Data Management for End-To-End Machine Learning, DEEM '18*, 2018. 16

[Par12] Aditya Parameswaran. An interview with surajit chaudhuri. In *XRDS: Crossroads, The ACM Magazine for Students, 19(1)*, pages 38–39, 2012. 5

[PGT] PGTune. https://pgtune.leopard.in.ua/. 25, 121

[Pos] PostgreSQL. http://www.postgresql.org/docs/9.4/static/release.html. 25

[SAC⁺79] P. Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979. 2, 3, 14, 22

[SDG17] Michael Shekelyan, Anton Dignös, and Johann Gamper. Digithist: a histogram-based data summary with tight error bounds. In *Proc. of the VLDB Endow., 10(11)*, pages 1514–1525, 2017. 15

[Ser] Microsoft SQL Server. Hints (transact-sql) - query. `https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-2017`. 127

[SLMK01] 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. 3

[TDJ13] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Efficiently adapting graphical models for selectivity estimation. In *Intl. Journal on Very Large Data Bases (VLDB), 22(1)*, pages 3–27, 2013. 6, 14

[Tin] Apache TinkerPop. `http://tinkerpop.apache.org/`. 141

[TK17] Immanuel Trummer and Christoph Koch. Solving the join ordering problem via mixed integer linear programming. In *Proc. of the 2017 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1025–1040, 2017. 20

[WBM+18] Florian Wolf, Michael Brendle, Norman May, Paul R. Willems, Kai-Uwe Sattler, and Michael Grossniklaus. Robustness metrics for relational query execution plans. In *Proc. of the VLDB Endow., 11(11)*, pages 1360–1372, 2018. 6, 14, 18, 129

[WCHN13] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. In *Proc. of the VLDB Endow., 6(10)*, pages 925–936, 2013. 24, 127, 135

[WCZ+13] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigumus, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Proc. of the 29th IEEE Intl. Conf. on Data Engg., ICDE '13*, pages 1081–1092, 2013. 20

[WDL09] Dingding Wang, Chris Ding, and Tao Li. K-subspace clustering. In *Proc.of the 2009 European Conf. on Machine Learning and Knowledge Discovery in Databases, ECML PKDD*, pages 506–521, 2009. 83

[WKG09] Janet L. Wiener, Harumi Kuno, and Goetz Graefe. Benchmarking query execution robustness. In *Proc. of the 1st TPC Technology Conference on Performance Evaluation and Benchmarking, TPCTC '09*, pages 153–166, 2009. 4

[WNS16] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. Sampling-based query re-optimization. In *Proc. of the 2016 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1721–1736, 2016. 17

[YHM15] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust query optimization methods with respect to estimation errors: A survey. In *ACM SIGMOD Record, 44(3)*, pages 25–36, 2015. 14

[ZPSP17] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. In *Proc. of the VLDB Endow., 10(8)*, pages 889–900, 2017. 19

9.A Query Text

> **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** d_year = 2001 **and** ss_list_price $<= 1.5$

Figure 9.1: 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** cs_sold_date_sk = d_date_sk **and** ca_gmt_offset = -7.0 **and** d_year = 1900 **and** cs_list_price $<=$ 10.5

Figure 9.2: Q15 (Based on TPC-DS Query 15)

> **select** i_item_id, ca_country, ca_state, ca_county, cs_quantity, cs_list_price, cs_coupon_amt, cs_sales_price, cs_net_profit, c_birth_year, cd1.cd_dep_count
> **from** catalog_sales, date_dim, item, customer_demographics cd1, customer, customer_demographics cd2, customer_address
> **where** cs_sold_date_sk = d_date_sk **and** cs_item_sk = i_item_sk **and** cs_bill_cdemo_sk = cd1.cd_demo_sk **and** cs_bill_customer_sk = c_customer_sk **and** c_current_cdemo_sk = cd2.cd_demo_sk **and** c_current_addr_sk = ca_address_sk **and** cd1.cd_gender = 'F' **and** cd1.cd_education_status = '2 yr Degree' **and** c_birth_month in (10, 9, 7, 5, 1, 3) **and** d_year = 2001 **and** ca_gmt_offset = -7 **and** i_current_price $<=$ 10

Figure 9.3: Q18 (Based on TPC-DS Query 18)

```
select i_brand_id, i_brand, i_manufact_id, i_manufact, ss_ext_sales_price
from store_sales, date_dim, item, customer, customer_address, store
where d_date_sk = ss_sold_date_sk and ss_item_sk = i_item_sk and ss_store_sk
= s_store_sk and ss_customer_sk = c_customer_sk and c_current_addr_sk =
ca_address_sk and i_manager_id=97 and d_moy=12 and d_year=2002 and
ss_list_price <= 17.5
```

Figure 9.4: Q19 (Based on TPC-DS Query 19)

```
select w_warehouse_name,i_item_id, sum(inv_quantity_on_hand)
from inventory, warehouse, item, date_dim
where i_item_sk = inv_item_sk and w_warehouse_sk = inv_warehouse_sk and
d_date_sk = inv_date_sk and i_current_price between 0.99 and 1.49 and d_date
between cast ('2002-01-26' as date) and cast ('2002-03-26' as date)
group by w_warehouse_name, i_item_id
```

Figure 9.5: Q21 (Based on TPC-DS Query 21)

```
select i_product_name, i_brand, i_class, i_category, sum(inv_quantity_on_hand)
from inventory, warehouse, item, date_dim
where i_item_sk = inv_item_sk and w_warehouse_sk = inv_warehouse_sk and
d_date_sk = inv_date_sk and d_year = 1998
group by i_product_name, i_brand, i_class, i_category
```

Figure 9.6: Q22 (Based on TPC-DS Query 22)

```
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 <= 10
group by i_item_id
order by i_item_id
```

Figure 9.7: Q26 (Based on TPC-DS Query 26)

```
select *
from store_sales, date_dim, item, store, customer_demographics
where ss_sold_date_sk = d_date_sk and ss_item_sk = i_item_sk and ss_store_sk
= s_store_sk and ss_cdemo_sk = cd_demo_sk and cd_gender = 'F' and
cd_marital_status = 'D' and cd_education_status = 'Primary' and d_year = 2000
and s_state in ('TN') and i_current_price <= 15
```

Figure 9.8: Q27 (Based on TPC-DS Query 27)

```
select  i_item_id,   i_item_desc,   s_store_id,   s_store_name,   ss_quantity,
sr_return_quantity, cs_quantity
from date_dim d1, date_dim d2, date_dim d3, store_sales, store_returns, cata-
log_sales, store, item
where d1.d_date_sk = ss_sold_date_sk and sr_returned_date_sk = d2.d_date_sk
and cs_sold_date_sk = d3.d_date_sk and s_store_sk = ss_store_sk and i_item_sk =
ss_item_sk and ss_customer_sk = sr_customer_sk and ss_item_sk = sr_item_sk and
ss_ticket_number = sr_ticket_number and sr_customer_sk = cs_bill_customer_sk
and sr_item_sk = cs_item_sk and d1.d_year = 1999 and d2.d_moy between 4 and
9 and d2.d_year = 2000 and d3.d_year in (2000,2001,2002,2003,2004,2005)
and i_current_price <= 20
```

Figure 9.9: Q36 (Based on TPC-DS Query 36)

```
select i_item_id, i_item_desc, i_current_price
from store_sales, date_dim, item, store
where d_date_sk = ss_sold_date_sk and i_item_sk = ss_item_sk and s_store_sk
= ss_store_sk and s_state in ('TN','TN','TN','TN','TN','TN','TN','TN') and d_year =
2001
group by i_item_id, i_item_desc, i_current_price
order by i_item_id
```

Figure 9.10: Q37 (Based on TPC-DS Query 37)

```
select w_state, i_item_id, sum(cs_sales_price)
from catalog_sales, warehouse, item, date_dim
where i_item_sk = cs_item_sk and w_warehouse_sk = cs_warehouse_sk and
d_date_sk = cs_sold_date_sk and i_current_price between 0.99 and 1.49 and
d_date between (cast ('2001-01-17' as date)) and (cast ('2001-03-17' as date))
group by w_state,i_item_id
order by w_state,i_item_id
```

Figure 9.11: Q40 (Based on TPC-DS Query 40)

```
select *
from item, store_sales, date_dim, store
where i_item_sk = ss_item_sk and d_date_sk = ss_sold_date_sk and s_store_sk =
ss_store_sk and d_year in (2001) and i_category in ('Books', 'Children', 'Electron-
ics') and i_class in ('personal', 'portable', 'reference', 'self-help') and i_brand in
('scholaramalgamalg #14', 'scholaramalgamalg #7', 'exportiunivamalg #9', 'schol-
aramalgamalg #9')
```

Figure 9.12: Q53 (Based on TPC-DS Query 53)

```
select sm_type, web_name, sum(case when (ws_ship_date_sk - ws_sold_date_sk
<= 30 ) then 1 else 0 end) as "30 days", sum(case when (ws_ship_date_sk -
ws_sold_date_sk > 30) and (ws_ship_date_sk - ws_sold_date_sk <= 60) then 1
else 0 end ) as "31-60 days", sum(case when (ws_ship_date_sk - ws_sold_date_sk
> 60) and (ws_ship_date_sk - ws_sold_date_sk <= 90) then 1 else 0 end) as
"61-90 days", sum(case when (ws_ship_date_sk - ws_sold_date_sk > 90) and
(ws_ship_date_sk - ws_sold_date_sk <= 120) then 1 else 0 end) as "91-120 days",
sum(case when (ws_ship_date_sk - ws_sold_date_sk > 120) then 1 else 0 end) as
">120 days"
from web_sales, warehouse, ship_mode, web_site, date_dim
where d_date_sk = ws_ship_date_sk and w_warehouse_sk = ws_warehouse_sk
and sm_ship_mode_sk = ws_ship_mode_sk and web_site_sk = ws_web_site_sk and
d_year = 2001
group by sm_type, web_name
order by sm_type, web_name
```

Figure 9.13: Q62 (Based on TPC-DS Query 62)

154

```
select i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy,
s_store_id, sum(coalesce(ss_sales_price*ss_quantity,0)) sumsales
from store_sales, date_dim, store, item
where d_date_sk = ss_sold_date_sk and i_item_sk = ss_item_sk and s_store_sk =
ss_store_sk and d_year=1999
group by i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy,
s_store_id
```

Figure 9.14: Q67 (Based on TPC-DS Query 67)

```
select ss_ticket_number, ss_customer_sk, count(*) cnt
from store_sales, date_dim, store, household_demographics
where d_date_sk = ss_sold_date_sk and s_store_sk = ss_store_sk and hd_demo_sk
= ss_hdemo_sk and d_dom <= 2 and hd_buy_potential = '> 10000' and
hd_vehicle_count > 0 and d_year in (2000, 2000+1, 2000+2) and s_county
in ('Williamson County', 'Williamson County', 'Williamson County', 'Williamson
County')
group by ss_ticket_number, ss_customer_sk
```

Figure 9.15: Q73 (Based on TPC-DS Query 73)

```
select c_customer_id as customer_id, c_last_name , c_first_name
from customer, customer_address, customer_demographics, house-
hold_demographics, income_band, store_returns
where c_current_addr_sk = ca_address_sk and cd_demo_sk = c_current_cdemo_sk
and hd_demo_sk = c_current_hdemo_sk and ib_income_band_sk =
hd_income_band_sk and sr_cdemo_sk = cd_demo_sk and ib_lower_bound
>= 52066 and ib_upper_bound <= 52066 + 50000 and ca_gmt_offset=-7
```

Figure 9.16: Q84 (Based on TPC-DS Query 84)

```
select i_category, i_class, i_brand, s_store_name, s_company_name, d_moy,
sum(ss_sales_price)
from item, store_sales, date_dim, store
where i_item_sk = ss_item_sk and d_date_sk = ss_sold_date_sk and s_store_sk =
ss_store_sk and d_year in (1999) and i_category in ('Jewelry', 'Electronics', 'Mu-
sic') and i_class in ('mens watch', 'wireless', 'classical')
group by i_category, i_class, i_brand, s_store_name, s_company_name, d_moy
```

Figure 9.17: Q89 (Based on TPC-DS Query 89)

```
select cc_call_center_id, cc_name, cc_manager, sum(cr_net_loss)
from call_center, catalog_returns, date_dim, customer, customer_address, cus-
tomer_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
order by sum(cr_net_loss)
desc
```

Figure 9.18: 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 = hd_demo_sk and
ss_store_sk = s_store_sk and t_hour = 8 and t_minute >= 30 and hd_dep_count =
2 and s_store_name = 'ese' and ss_list_price <= 19.5
```

Figure 9.19: Q96 (Based on TPC-DS Query 96)

```
select sm_type, cc_name, count(*)
from catalog_sales, warehouse, ship_mode, call_center, date_dim
where d_date_sk = cs_ship_date_sk and w_warehouse_sk = cs_warehouse_sk and
sm_ship_mode_sk = cs_ship_mode_sk and cc_call_center_sk = cs_call_center_sk
and d_year = 2002
group by sm_type, cc_name
order by sm_type, cc_name
```

Figure 9.20: Q99 (Based on TPC-DS Query 99)