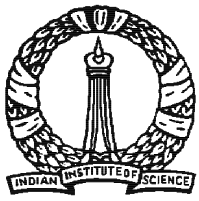
Efficiently Approximating Query Optimizer Diagrams

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

By
Atreyee Dey



Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

August 2009

Acknowledgements

At the outset, I would like to convey my profound regard and sincere thanks to my advisor Prof. Jayant Haritsa, for giving me an opportunity to work under him and feel myself quite fortunate on being supervised by him. In spite of very busy schedule he did not hesitate to spend his valuable time and supported me throughout my research work with his patience and knowledge. I owe to my advisor for teaching me what research is all about.

Also, I do convey my heartfelt gratitude to Sourjya and Harish for their valuable contributions and suggestions in structuring the Joint Paper, as well as my understanding of query optimization and research in general. The same is true with Abhijit who helped me generously during my initial days at the lab and continued to help during the year that we shared together. I enjoyed the company of a friendly and cheerful group of fellow lab-mates Abhirama, Harsh, Rajul and Ravi. I do solemnly, acknowledge their enthusiastic support to review part of my thesis.

I specially thank Mr. Shekhar, Mr. Madan and Ms. Mallika who were constantly there on my behalf to sort out the administration issues those came up at times and spared me the agonies of wasting time and effort on them.

I do earnestly convey my thanks to all the supporting staffs of the Institute for their cooperation and help extended to me during my entire stay in the lab and campus.

Finally, my deepest thanks are to my beloved parents, whose affection and support make everything possible; they are my backbone in guiding me eternally in the right direction.

Abstract

Modern database systems use a query optimizer to identify the most efficient strategy, called "query execution plan", to execute declarative SQL queries. The role of the query optimizer is especially critical for the complex decision-support queries featured in current data warehousing and data mining applications.

Given an SQL query template that is parametrized on the selectivities of the participating base relations and a choice of query optimizer, a *plan diagram* is a color-coded pictorial enumeration of the execution plan choices of the optimizer over the query parameter space. Complementary to the plan-diagrams are *cost* and *cardinality diagrams* which graphically plot the estimated execution costs and cardinalities respectively, over the query parameter space. These diagrams are collectively known as *optimizer diagrams*. Optimizer diagrams have proved to be a powerful tool for the analysis and redesign of modern optimizers, and are gaining interest in diverse industrial and academic institutions. However, their utility is adversely impacted by the impractically large computational overheads incurred when standard brute-force approaches are used for producing fine-grained diagrams on high-dimensional query templates.

In this thesis, we investigate strategies for efficiently producing close approximations to complex optimizer diagrams. Our techniques are customized for different classes of optimizers, ranging from the generic Class I optimizers that provide only the optimal plan for a query, to Class II optimizers that also support costing of sub-optimal plans and Class III optimizers which offer enumerated rank-ordered lists of plans in addition to both the former features.

For approximating plan diagrams for Class I optimizers, we first present database-

oblivious techniques based on classical *random sampling* in conjunction with nearest-neighbor (NN) inference scheme. Next we propose *grid sampling* algorithms which consider database specific knowledge such as (a) the structural differences between the operator trees of plans on the grid locations and (b) parametric query optimization principle. These algorithms become more efficient when modified to exploit the sub-optimal plan costing feature available with Class II optimizers. The final algorithm developed for Class III optimizers assume plan cost monotonicity and utilize the rank-ordered lists of plans to efficiently generate *completely accurate* optimizer diagrams. Subsequently, we provide a relaxed variant, which trades quality of approximation, for reduction in diagram generation overhead. Our proposed algorithms are capable of terminating according to user given error bounds for plan diagram approximation.

For approximating cost diagrams, our strategy is based on linear least square regression performed on a mathematical model of plan cost behavior over the parameter space, in conjunction with interpolation techniques. Game theoretic and linear programming approaches have been employed to further reduce the errors in cost approximation.

For approximating cardinality diagrams, we propose a novel parametrized mathematical model as a function of selectivities for characterizing query cardinality behavior. The complete cardinality model is constructed by clustering the data points according to their cardinality values and subsequently fitting the model through linear least square regression technique separately for each cluster. For non-sampled data points the cardinality values are estimated by first determining the cluster they belong to and then interpolating the cardinality value according to the suitable model.

Extensive experimentation with a representative set of TPC-H and TPC-DS-based query templates on industrial-strength optimizers indicates that our techniques are capable of delivering 90% accurate optimizer diagrams while incurring no more than 20% of the computational overheads of the exhaustive approach. In fact, for full-featured optimizers, we can guarantee zero error optimizer diagrams which usually require less than 10% overheads. Our results exhibit that (a) the approximation is materially faithful to the features of the exact optimizer diagram, with the errors thinly spread across the pic-

ture and largely confined to the plan transition boundaries and (b) the cost increase at the non-sampled point due to assignment of sub-optimal plan is also limited.

These approximation techniques have been implemented in the publicly available Picasso optimizer visualizer tool. We have also modified PostgreSQL's optimizer to incorporate costing of sub-optimal plans and enumerating rank-ordered lists of plans. In addition to these, we have designed estimators for predicting the time overhead involved in approximating optimizer diagrams with regard to user given error bounds.

In summary, this thesis demonstrates that accurate approximations to exact optimizer diagrams can indeed be obtained cheaply and consistently, with typical overheads being an order of magnitude lower than the brute-force approach. We hope that our results will encourage database vendors to incorporate the foreign-plan-costing and plan-rank-list features in their optimizer APIs.

Publications

1. Atreyee Dey, Sourjya Bhaumik, Harish D. and Jayant Haritsa,
   "*Efficiently Approximating Query Optimizer Plan Diagrams*",
   Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB), pages 1325-1336
   Auckland, New Zealand, August, 2008

2. Atreyee Dey, Sourjya Bhaumik, Harish D. and Jayant Haritsa,
   "*Efficient Generation of Approximate Plan Diagrams*",
   Technical Report, TR-2008-01, DSL/SERC, Indian Institute of Science,
   http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2008-01.pdf

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The Structured Query Language (SQL) [58] is the international standard for querying relational database management systems (DBMS) such as IBM's DB2, Microsoft's SQL Server, Oracle etc., which form the cornerstone of today's information industry. SQL is a declarative language in the sense that an SQL query specifies *what* has to be done, not *how* it is to be done. A sample SQL query on the TPC-H benchmark schema [83] is given in Figure 1.1, which lists the mode of shipping for all items whose quantity is less than or equal to 20, and forms a part of an order of price 100 or less.

```
select l_shipmode
from orders, lineitem
where o_orderkey = l_orderkey
        and o_totalprice ≤ 100
        and l_quantity ≤ 20
group by l_shipmode
order by l_shipmode
```

Figure 1.1: Sample SQL Query

Modern database systems use a *query optimizer* to identify the most efficient strategy to execute declarative SQL queries. The efficiency of the strategies, called "query execution plans" or simply "plans", is usually evaluated or costed in terms of the estimated

Figure 1.2: Query Execution Plan

query response time. A sample plan for the query of Figure 1.1 is shown in Figure 1.2. This plan performs sequential scans of the ORDERS and LINEITEM relations before joining them using the *hash join* operator. It finally sorts and groups the results in the required order.

Optimization is a mandatory exercise since the difference between the cost of the best plan and a random choice could be in orders of magnitude [73]. The role of query optimizers has become especially critical in recent times due to the high degree of query complexity characterizing current decision-support applications, as exemplified by the TPC-H benchmark [83], and its new incarnation, TPC-DS [82].

Query optimization is a difficult problem due to the large number of possible ways to execute a given query using different access methods, join orders, join operators, etc. While industrial strength query optimizers each have their own proprietary methods to identify the best plan, the de-facto standard underlying strategy is based on the classical System-R optimizer [68] proposed about three decades ago. This method is: Given a user query, first apply a variety of heuristics to restrict the combinatorially large search space of plan alternatives to a manageable size; then estimate, with a cost model and a dynamic-programming-based processing algorithm, the efficiency of each of these candidate plans;

finally, choose the plan with the lowest estimated cost.

Query optimization using this cost-based approach is computationally expensive w.r.t. the time and resources that need to be expended to find the best plan. Therefore, understanding and characterizing query optimizers with the ultimate objective of improving their performance is a fundamentally important issue in the database research literature.

1.1 Query Templates

select o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume)

from (

select year(o_orderdate) as o_year, l_extendedprice * (1 - l_discount) as volume,
n2.n_name as nation

from part, supplier, lineitem, orders, customer, nation n1, nation n2, region

where p_partkey = l_partkey and s_suppkey = l_suppkey and l_orderkey = o_orderkey
and o_custkey = c_custkey and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey and s_nationkey = n2.n_nationkey
and r_name = 'AMERICA' and p_type = 'ECONOMY ANODIZED STEEL'
and **s_acctbal** $\leq C_1$ and **l_extendedprice** $\leq C_2$

) as all_nations

group by o_year

order by o_year

Figure 1.3: Example Query Template: QT8

The cost of a given query execution plan is a function of many parameters, including the database structure and contents, the engine settings, the system configuration, etc. For a query on a given database and system configuration, the optimizer's plan choice is primarily a function of the *selectivities* of the base relations participating in the query – that is, the estimated number of rows of each relation relevant to producing the final result. Varying the selectivities of one or more of the base relations produces the selectivity space w.r.t. these relations. A "parameterized query template" is a query with predicates that produce queries across this selectivity space.

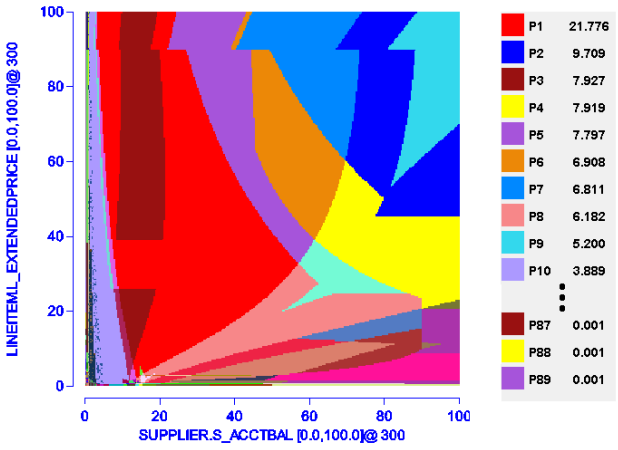
For example, consider QT8, the parameterized 2-D query template shown in Figure 1.3, based on Query 8 of the TPC-H benchmark, with selectivity variations on the SUPPLIER and LINEITEM relations through the s_acctbal $\leq C_1$ and l_extendedprice $\leq C_2$ predicates, respectively. By varying the constants C_1 and C_2, queries are generated across the selectivity space.

1.2 *Optimizer Diagrams*

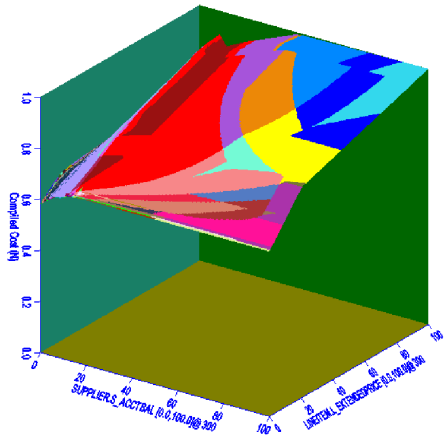
The behavior of a query optimizer over the selectivity space can be captured in a suite of diagrams. First, a "plan diagram" [64] denotes a color-coded pictorial enumeration of the execution plan choices of a database query optimizer for a parameterized query template over the relational selectivity space. The plan diagram for QT8 (produced using the Picasso optimizer visualization tool [76] on a popular commercial database engine) is shown in Figure 1.4(a), where the X and Y axes determine the percentage selectivities of the SUPPLIER and LINEITEM relations, respectively, and each color-coded region represents a particular plan that has been determined by the optimizer to be the optimal choice in that region. We find that a set of 89 different optimal plans, P1 through P89, cover the entire selectivity space. The value associated with each plan in the legend indicates the percentage area coverage of that plan in the diagram – P1, for example, covers about 22% of the space, whereas P89 is chosen in only 0.001% of the space. ***[Note to Readers: We recommend viewing all diagrams presented in this thesis directly from the color PDF file, available at http://dsl.serc.iisc.ernet.in/ atreyee/thesis_draft.pdf, or from a color print copy, since the greyscale version may not clearly register the various features.]***

Complementary to the plan diagram is a "cost diagram", shown in Figure 1.4(b), which is a three-dimensional visualization of the estimated plan execution costs over the same relational selectivity space. The X and Y axes represent the variations in selectivities and the Z axis represents the cost. In this picture, the costs are normalized to the maximum cost over the space, which in this case is 155 and occurs at the point corresponding to maximum selectivity along the X and Y axes. The minimum and maximum estimated

costs are also shown in the bottom rectangle of the diagram.

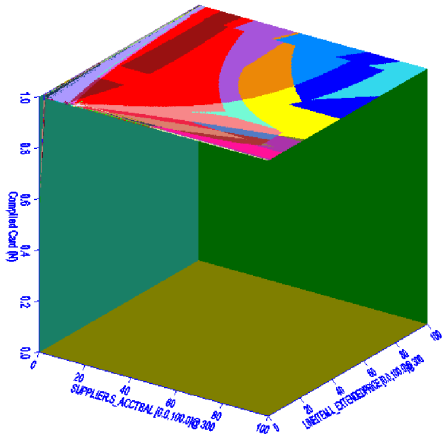


(a) **Plan Diagram**



Min Est Cost : 9.04E1
Max Est Cost : 1.55E2

(b) **Cost Diagram**



Min Est Card : 3.74E0
Max Est Card : 7.00E0

(c) **Cardinality Diagram**

Figure 1.4: Sample Optimizer Diagrams (QT8)

Finally, a "cardinality diagram", shown in Figure 1.4(c), is similar to a cost diagram except that it shows the cardinality of the query result as estimated by the optimizer, instead of execution cost. The minimum and maximum estimated cardinalities are also shown in the bottom rectangle. From here onwards we will use the term *Optimizer diagrams* to collectively refer to plan, cost and cardinality diagrams.

1.3 Applications of *Optimizer Diagrams*

Since their introduction in [64] a few years ago, Query *Optimizer Diagrams* i.e. plan, cost and cardinality diagrams have proved to be a powerful tool for the analysis and redesign of industrial-strength database query optimizers. For example, as evident from Figure 1.4(a), plan diagrams can be surprisingly complex and dense, with a large number of plans covering the space – several such instances spanning a representative set of benchmark-based query templates on current optimizers are available at [76]. Our interactions with industrial development teams have indicated that these diagrams have often proved to be contrary to the prevailing conventional wisdom. The reason is that while optimizer behavior on *individual queries* has certainly been analyzed extensively by developers, plan diagrams provide a completely different perspective of behavior *over an entire space*, vividly capturing plan transition boundaries and optimality geometries. So, in a literal sense, they deliver the "big picture". Similarly the cost and cardinality diagrams shown in 1.4(b) and 1.4(c), help in visualizing the estimations generated by the optimizer.

Optimizer diagrams are currently in use at various industrial and academic sites for a diverse set of applications including analysis of existing optimizer designs; visually carrying out optimizer regression testing; debugging new query processing features; comparing the behavior between successive optimizer versions; investigating the structural differences between neighboring plans in the space; evaluating the variations in the plan choices made by competing optimizers; etc. As a case in point, visual examples of non-monotonic cost behavior in commercial optimizers, indicative of modeling errors, were highlighted in [64].

Apart from optimizer design support, plan diagrams can also be used in operational settings. Specifically, since they identify the optimal set of plans for the entire relational selectivity space at compile-time, they can be used at run-time to immediately identify the best plan for the current query without going through the time-consuming optimization exercise. Further, they can prove useful to adaptive plan selection techniques (e.g. [17, 23, 57]) which, based on the differences between the actual selectivities encountered during execution and the associated compile-time estimates, may dynamically choose to re-optimize the query and switch plans mid-way through the processing.

In this context, plan diagrams can help to eliminate the re-optimization overheads incurred in determining the substitute plan choices for estimation errors that occur on the selectivity dimensions.

Plan Diagram Reduction. A particularly compelling utility of plan diagrams is that they provide the input to "plan diagram reduction" algorithms. Specifically, given a plan diagram and a cost-increase-threshold (λ) specified by the user, these reduction algorithms recolor the dense diagram to a simpler picture that features only a subset of the original plans while ensuring that the cost of no individual query point goes up by more than λ percent, relative to its original cost. That is, some of the original plans are "completely swallowed" by their siblings, leading to a reduced plan cardinality in the diagram. It has been shown [18] that if users were willing to tolerate a minor cost increase of $\lambda = 20\%$, the absolute number of plans in the final reduced picture could be brought down to *within or around ten*. In short, that complex plan diagrams can be made "anorexic" while retaining acceptable query processing performance. For example, the reduced version of the QT8 plan diagram (Figure 1.4(a)) retains *only 7* of the original 89 plans with $\lambda = 20\%$ as shown in Figure 1.5.

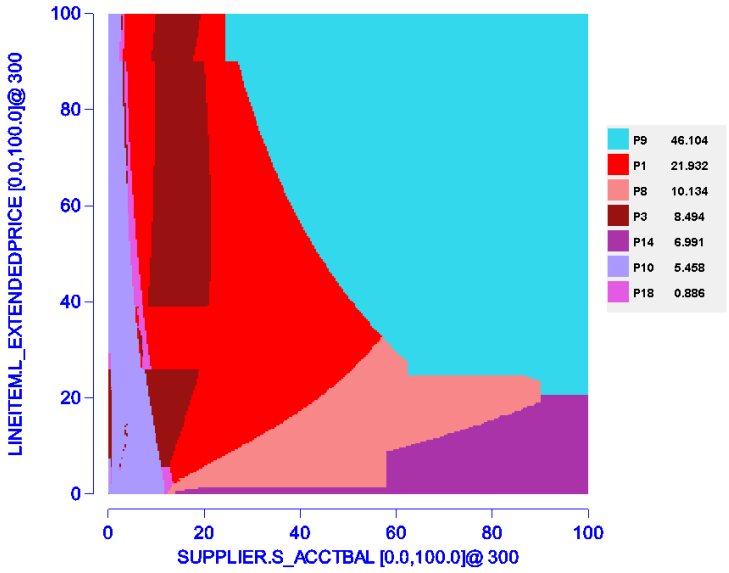


Figure 1.5: Reduced Diagram ($\lambda = 20\%$)

Anorexic plan diagram reduction has significant practical benefits [18], including quantifying the redundancy in the plan search space, enhancing the applicability of parametric query optimization (PQO) techniques [47, 48], identifying error-resistant and least-expected-cost plans [14, 15], and minimizing the overhead of multi-plan approaches [3, 52]. A detailed study of its application to identifying robust plans that are resistant to errors in relational selectivity estimates is available in [19].

1.4 Generation of *Optimizer Diagrams*

The generation and analysis of *Optimizer Diagrams* has been facilitated by the development of the Picasso optimizer visualization tool [76]. Given a multi-dimensional SQL query template like QT8 and a choice of database engine, the Picasso tool automatically produces the associated plan, cost and cardinality diagrams. It is operational on several major platforms including IBM DB2, Oracle, Microsoft SQL Server, Sybase ASE and PostgreSQL.

The diagram production strategy used in Picasso is the following:

Plan Diagram Given a d-dimensional query template and a plot resolution of r, the Picasso tool generates r^d queries that are either uniformly or exponentially (user's choice) distributed over the selectivity space. Then, for each of these query points, based on the associated selectivity values, a query with the appropriate constants instantiated is submitted to the query optimizer to be "explained" – that is, to have its optimal plan computed. After the plans corresponding to all the points are obtained, a different color is associated with each unique plan, and all query points are colored with their associated plan colors. Then, the rest of the diagram is colored by painting the region around each point with the color corresponding to its plan. For example, in a 2D plan diagram with a uniform grid resolution of 10, there are 100 real query points, and around each such point a square of dimension 10x10 is painted with the point's associated plan color.

Cost and Cardinality Diagrams Each of these r^d queries on being explained also gets

assigned with the estimated cost and cardinality values. The plot of the cost and cardinality values normalized to the maximum cost and cardinality values in the set of r^d queries respectively are known as Cost and Cardinality diagrams. Given a d-dimensional query template, the cost and cardinality diagrams are each a $d+1$ dimensional diagram.

The above exhaustive approach is eminently acceptable for diagrams on low-dimension (1D and 2D) query templates with coarse resolutions (up to 100 points per dimension). However, it becomes impractically expensive for higher dimensions and fine-grained resolutions due to the exponential growth in overheads. For example, a 2D optimizer diagram with a resolution of 1000 on each selectivity dimension, or a 3D optimizer diagram with a resolution of 100 on each dimension, both require invoking the optimizer a *million* times. Even with a conservative estimate of about half-second per optimization, the total time required to produce the picture is close to a week! Therefore, although *Optimizer Diagrams* have proved to be extremely useful, their high-dimension and/or fine-resolution versions pose serious computational challenges.

In this thesis we address the problem of efficiently generating optimizer diagrams. Two obvious mechanisms to lower the computational time overheads are:

1. Customize the resolution on each dimension to be domain-specific – for example, coarse resolutions may prove sufficient for categorical data

2. Use computational units in parallel to leverage the independence between the optimizations of the individual query points, resulting in concurrent issue of multiple optimization requests

In this thesis, we consider how we can supplement the above remedies, which may not always be applicable or feasible, through the use of generic *algorithmic* techniques, as described next.

1.5 Motivation and Research Challenges

Specifically, we investigate whether it is possible to efficiently produce *accurate approximations* to plan, cost and cardinality diagrams.Denoting the exact plan diagram as P and the approximation as A, there are two categories of errors that arise in the process for approximating plan diagram:

Plan Identity Error (ϵ_I): It refers to the possibility of the approximation missing out on a subset of the plans present in the exact plan diagram. It is computed as the percentage of plans lost in A relative to P.

Plan Location Error (ϵ_L): It refers to the possibility of incorrectly assigning plans to query points in the approximate plan diagram. It is computed as the percentage of incorrectly (relative to P) assigned points in A.

Similarly, errors associated with the cost and cardinality diagram approximations are:

Maximum Error (ϵ_{MAX}): It refers to the maximum percentage cost/cardinality error incurred in A with regard to P, measured over all points in the diagram.

Root Mean Square Error (ϵ_{RMS}): It refers to the root mean square error caused by the cost/cardinality approximation. RMS error indicates the average quality of the process.

The approximation strategies should be robust enough to counteract the following challenges:

- The ϵ_I error is difficult to control since a majority of the plans appearing in plan diagrams, as seen in Figure 1.4(a), are very small in area, and therefore hard to find. Similarly the ϵ_L error is also hard to control since the plan boundaries, as seen in Figure 1.4(a), can be highly non-linear, and are sometimes even irregular in shape [76].

- Cost being a function of a plan structure needs to be estimated separately for individual plans, which will be subsequently helpful in approximating cost value

for un-optimized points. However, scarcity of sufficient number of data points for small plans severely affects the quality of the respective cost functions. Moreover, extrapolation error also imposes a significant threat to the quality of approximate cost diagram.

- In approximating cardinality diagram our main challenge was to come up with a suitable model for depicting cardinality diagram first and then use it for cardinality value estimation.

We have observed that some of the advanced commercial optimizers provide extra features other than the best plan, which can improve the approximation efficiency. As described next, these lead us to categorize the current optimizers into different classes and devise separate strategies tailored to their capabilities.

1.5.1 Optimizer Classes

Our study shows that the ability to reduce overheads is a function of the plan-related functionalities offered by the optimizer's API, based on which we define the following three categories of optimizers:

Class I: OP Optimizers This class refers to the generic cost-based optimizers that are routinely found in virtually every enterprise database product, where the API only provides the optimal plan (OP), as determined by the optimizer, for a user query.

Class II: OP + FPC Optimizers This class of optimizers additionally provide a "foreign plan costing" (FPC) feature in their API, that is, of costing plans *outside* their native optimality regions. Specifically, the feature supports the following "what-if" question: "What is the estimated cost of sub-optimal plan p if utilized at query location q?". FPC has become available in the current versions of several industrial-strength optimizers, including DB2 [80] (Optimization Profile), SQL Server [81] (XML Plan), and Sybase [78] (Abstract Plan). Note that along with cost, the FPC feature also returns the estimated result cardinality of the query point.

Class III: OP + FPC + PRL Optimizers This class of optimizers support, in addition to FPC, an API that provides not just the best plan but a "plan-rank-list" (PRL), enumerating the top k plans for the query. For example, with $k = 2$, both the best plan and the second-best plan are obtained when the optimizer is invoked on a query. The PRL feature can be implemented in current optimizers through modifying the Dynamic Programming-based query optimization process. However, to our knowledge, it is not yet available in any of the current systems. Therefore, we showcase its utility through our own implementation in a public-domain optimizer. Note that conceptualization and implementation of algorithms to extract PRL was another challenging task – the details of which are given later in Section 3.4.1.

1.5.2 Approximation Techniques

We first concentrate on generating an accurate approximation of plan diagram, which requires optimizing a sub-set of points from the set of r^d query points. Subsequently the approximate cost and cardinality diagrams are produced, without involving any further optimizations. For Class I (OP) and Class II (OP+FPC) optimizers, the techniques that we propose for plan diagram approximation are based on a combination of sampling and inference, while for Class III optimizers (OP+FPC+PRL), it is purely based on inference. The sampling techniques include both classical random sampling and grid sampling, while the inference approaches rely on nearest-neighbor (NN) classifiers [72], parametric query optimization (PQO) [47, 48] and plan cost monotonicity (PCM) [18]. For some of the techniques, theoretical results that help to provide guaranteed bounds on the errors are available, whereas for the others, empirical evaluation is the only recourse.

For Class I diagrams, the cost diagram approximation uses the plan cost model defined in [19]. The cost functions for individual plans discovered in A are determined using the linear least square regression technique. The derived plan cost functions are then used to estimate cost values at the inferred points. The interpolation errors due to inadequate number of data points is resolved by applying a game theoretic approach. The extrapolation error is addressed by restricting the model-based interpolation to the points

residing inside the convex polytope constructed by the optimized data points. We apply linear programming to identify the interior points. For points outside the polytope we apply linear extrapolation technique. Further, in this thesis we also propose a cardinality model as a function of relational selectivities. Fitting of such model requires application of a clustering algorithm beside the linear least square regression technique. Note that we may bypass cost and cardinality diagram approximation for Class II optimizers, since cost and cardinality values can be obtained using the FPC feature by costing each and every inferred point explicitly. Moreover, for Class III we will show that the exact cost and cardinality diagrams come for free as an output of the algorithms used for plan diagram approximation.

1.6 Results of the Speedup Quality

We have quantitatively assessed the efficacy of the various strategies, with regard to plan identity and location errors, through extensive experimentation with a representative suite of multi-dimensional TPC-H [83] and TPC-DS [82] based query templates on leading commercial and public-domain optimizers. We have also observed that the maximum and RMS errors caused due to the cost and cardinality approximations are low. Our results are very promising since they indicate that accurate approximations can indeed be obtained *cheaply and consistently*, as described below.

10 percent Error Bound. Consider the case where the user desires that the approximation error is of the order of *10 percent* or less on both plan identities and plan locations. For Class I (OP) optimizers, it is possible to regularly achieve this target with *only around 15% overheads* of the brute-force exhaustive method. To put this in perspective, the earlier-mentioned one-week plan diagram can be produced in less than a day. A sample approximate diagram (having 10% identity and 10% location error) is shown in Figure 1.6, with all the erroneous locations marked in black – as can be seen, the approximation is materially faithful to the features of the exact plan diagram, with the errors thinly spread across the picture and largely confined to the plan transition boundaries.

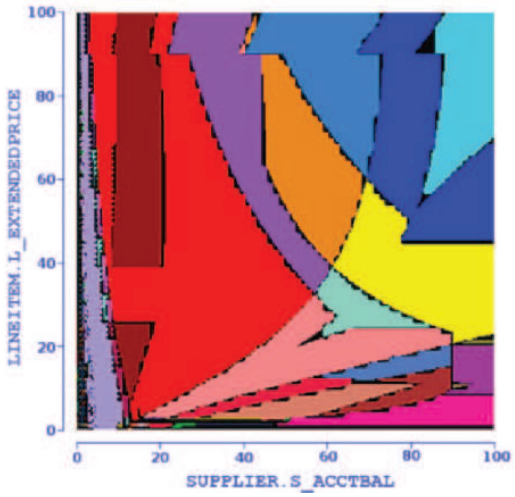


Figure 1.6: Approximate Diagram (10% Error Bound) - QT8

Database	Bound	Overhead %		ϵ_I %		ϵ_L %		Cost ϵ_{RMS} %		Card ϵ_{RMS} %	
	$\epsilon_I = \epsilon_L(\%)$	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg
TPC-H	1	40	16	1.3	0.5	1.1	0.3	1.5	0.2	8	2.2
	5	18.9	10.4	6	1	5.7	3.8	1.7	0.5	9.7	3.8
	10	14.8	7.1	10	2	10.1	6.7	2.9	0.6	11.3	3.7
	15	11.3	5.3	14	4	13.3	8.7	3.3	1.1	12.0	3.9
	20	8.6	4.5	19	6.5	19.5	12.5	8.6	1.4	14.9	4.3
TPC-DS	1	42	16	1.3	0.5	1.1	0.4	1.5	0.2	10	4.5
	5	22.2	10.1	6.1	1.5	5.8	3.4	1.7	0.5	12	6.1
	10	13.9	7.6	10.2	3.4	10.1	6.7	3.9	0.7	15	8.3
	15	12.8	6.2	15.6	5.3	12.7	8.5	7.1	1.4	15	8.6
	20	9.5	4.7	17.9	6.2	21.6	12.5	10	2.9	20	9

Table 1.1: Results for Optimizer Diagram Approximation for Class I

The approximate cost and cardinality diagrams associated with these 90% accurate plan diagrams incur very low value of RMS error e.g. within the range of 2-10%.

For Class II (OP+FPC) optimizers, a similar error performance is achieved with *only around 10% overheads*. An important point to note here is that plan costing is considerably cheaper than searching for the optimal plan. Finally, for Class III (OP+FPC+PRL) optimizers, the overheads come down to *less than 5%*. We can also obtain the associated exact cost and cardinality diagrams utilizing the FPC API. However, for the Class I optimizers due to the presence of location error the cost diagram may not be exact. As shown in Table 1.3 with 90% accurate plan diagram the cost penalty is below 5%.

1 percent Error Bound. We have also investigated the scenario where the user has the extremely stringent expectation of around *1 percent* plan identity and location errors. For this situation, Class I and II both take up to *40% overheads*, while Class III usually incurs less than *10% overheads*. For Class I and II optimizers the RMS error of the cost and cardinality approximation improves around 2% - 6% (refer to Tables 1.1 and 1.3).

Table 1.1 consolidates the approximation quality achieved for Class I optimizers, the generic optimizer with least features available for end-users. Experiments were performed on extensive set of TPC-H and TPC-DS benchmark query templates for varied resolutions (30 to 1000) and dimensions ($1D$ to $4D$). The "Max" and "Avg" columns depict the maximum and average values acquired by the associated parameters e.g. "Max" and "Avg" column of "Cost ϵ_{RMS} %" refers to the maximum and average RMS error incurred respectively by the cost approximation. This table also shows that without any extra feature we can achieve a speedup in order of magnitude. Results observed for Class II and III are at least twice as better than Class I.

1.7 Contributions

In this thesis we present a suite of algorithms customized to the optimizer's API richness, for efficiently generating accurate approximate *Optimizer Diagrams*. For plan diagram approximation we will first present the database-oblivious techniques based on classical random sampling which perform moderately in approximating the diagrams. These when further modified to consider database specific knowledge lowered the overhead by an order of magnitude. In this thesis, we also present an algorithm to generate exact plan diagrams with much less number of optimizations.

Considering the approximate plan diagram as an input we then move on to presenting cost and cardinality diagram approximation. We applied the cost model developed in [19] to first evaluate the cost function for individual plan through regression. These cost functions are then applied to estimate the cost values for inferred points. Similar to the process adopted in [19] we present the cardinality model as a function of selectivity

Optimizer Class	**Overheads Range (Bound : $\epsilon_I = \epsilon_L = 10\%$)**		**Overheads Range (Bound :$\epsilon_I = \epsilon_L = 1\%$)**	
	Plan	Cost & Card	Plan	Cost & Card
Class I (OP)	1% – 15%	0.02% – 0.05%	15% – 40%	0.01% – 0.02%
Class II (OP+FPC)	1% – 10%	6 – 10%	15% – 40%	3 – 7%
Class III (OP+FPC+PRL)	1% –5%	0%	1% – 10%	0%

Table 1.2: Summary Results for Optimizer Diagram Approximation

Optimizer Class	**(Bound = 10%)**		**(Bound = 1%)**	
	Cost ϵ_{RMS}	Card ϵ_{RMS}	Cost ϵ_{RMS}	Card ϵ_{RMS}
Class I (OP)	1-10%	0-15%	0-2%	0-5%
Class II (OP+FPC)	3%	0%	1%	0%
Class III (OP+FPC+PRL)	0%	0%	0%	0%

Table 1.3: Summary Results for Cost and Cardinality Errors

applicable for most of the modern optimizers available. This cardinality model is used afterwards to interpolate cardinality values for the inferred points following a similar approach used for cost approximation.

Results for *Optimizer Diagrams* (with uniform query point distribution) are summarized in Tables 1.2 and 1.3. In Table 1.2 the typical range of overheads (relative to the exhaustive approach) to generate the approximate *Optimizer Diagrams* is shown as a function of the user's error bound for each optimizer class. Finally in Table 1.3 we show the maximum RMS error incurred by the cost and cardinality approximation, taken over an extensive set of query templates of TPC-H and TPC-DS benchmarks. Note that the cost and cardinality errors are all 0 for Class III. However, even with the ability to cost each point explicitly, we could not achieve zero error cost diagram for Class II optimizers due to the presence of location error. At those points we actually cost a sub-optimal plan through FPC, which may be higher than the exact cost value. Whereas, the cardinality model obtained for Class II is perfect, since cardinality is not a function of the plans assigned at the inferred points.

1.8 Organization

The remainder of this thesis is organized as follows: Related works are reviewed in Chapter 2. The plan diagram approximation algorithms developed for the different classes of optimizers are presented in Chapter 3, which also contains the experimental results associated with each process. The cost value approximation techniques are illustrated in Chapter 4. Chapter 5 starts by presenting the cardinality model and later shows how to apply the same model for cardinality value approximation. Implementation details are discussed in Chapter 6. Finally, in Chapter 7, we summarize our conclusions and outline future research avenues.

Chapter 2

Survey of Related Research

There has been extensive work in the area of query optimization for relational database management systems since the early 70's, triggered by the advent of declarative query languages. A number of surveys (eg. [38, 13]) have covered the progress of query optimization techniques over the years. In this chapter, we also discuss some of the approximation techniques where classical random sampling approach has been exploited. Then we talk about some of the estimators devised for counting distinct values of an attribute in a relation. After this we examine research works done towards analyzing the industrial optimizers. We assume the reader is familiar with the techniques they discuss and only give a brief overview of the basic concepts here. Sections 2.1 and 2.4 in this chapter are reproduced verbatim from [20].

2.1 Challenges of Query Optimization

Among the key constituents of the query evaluation component of an SQL database system are the *query optimizer* and the *query execution engine*. The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of an SQL query as input and is responsible for generating an efficient execution plan for the given SQL query from the space of possible execution plans. One aspect of optimization is where the system attempts to find an expression equivalent to the given expression, but

more efficient to execute. Another aspect is selecting a detailed strategy for processing the query. The task of an optimizer is computationally challenging since, for a given SQL query, there can be a large number of possible execution plans, of the order of $O(3^n)$ [71], where n is the number of base relations in the query. The query execution engine implements the set of physical operators specified by the execution plan. Each operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are sequential scan, index scan, (external) sort, nested-loop join and sort-merge join.

The design of a query optimizer entails tackling the following challenging issues:

2.1.1 Plan Selection Strategy

A number of selection strategies can be applied for query optimization. These include:

1. Make a random choice.
2. Use a set of heuristic rules.
3. Use randomized algorithms or genetic techniques.
4. Exhaustively enumerate the search space and use a cost-based approach.

The exhaustive cost-based approach mentioned in 4 is the most commonly used in modern optimizers, since none of the others can deterministically guarantee the quality of their choices. Note that the approaches mentioned in 2 and 3 could be cost-based too. The pioneering work in the development of cost-based optimizers was carried out in the System-R project [68]. Their techniques have been incorporated in many commercial optimizers and continue to be remarkably relevant. In System-R, the size of the search space is restricted by considering only the set of left-deep plans, which allows pipelining of the output of one operator to the input of the next operator. It also introduced the notion of "interesting orders" into the plan selection process – the ordering of the output tuples of an operator is called an interesting order if it can become useful in some subsequent operation. The idea of an interesting order was later generalized to *physical properties* [36], which refers

to any characteristic of a plan that is not necessarily shared by other plans for the same expression, but could impact the cost of subsequent operations.

In an alternative strategy, Chu and Halpern [15, 14] propose the idea of picking the *least expected cost* (LEC) plan rather than the least specific cost (LSC) plan. In their first paper [15], they propose a set of algorithms to find this LEC plan and guarantee that this plan will be at least as good as the LSC plan, and typically better. They also consider parameters that could vary during the execution of the plan. They find that "*The greater the run-time variation in the values of parameters that affect the cost of the query plan, the greater the cost advantage of the LEC plan is likely to be*". They assume that the probability distribution of the values of the parameters is available at compile-time.

In their second paper [14], they observe that the LSC optimization does, in many cases, yield the LEC plan. The current optimizers can be coaxed to pick the LEC plan by appropriately choosing the parameters and their settings. They also study cases where running time is not the cost measure applied (it may matter if the plan is blocking or produces results at a constant rate, etc.) and find that in these scenarios, LEC optimization becomes particularly relevant.

2.1.2 Efficient Selection Strategies

For the cost-based optimizers, System-R proposed the use of dynamic programming to efficiently find a good plan. The dynamic programming approach is based on the assumption of the *principle of optimality* [77], which states that the optimal solution to a problem is a combination of optimal solutions to its subproblems. While dynamic programming (DP) works very well for moderately complex queries with up to around a dozen base relations, it usually fails to scale beyond this stage in current systems due to its inherent exponential space and time complexity. Therefore, DP becomes practically infeasible for complex queries with a large number of base relations.

To address the above problem, a variety of approaches have been proposed in the literature, such as Iterative Dynamic Programming (IDP) [54, 69], wherein DP is employed bottom-up until it hits its feasibility limit, and then iteratively restarted with a

significantly reduced subset of the execution plans currently under consideration. A recent alternative approach that improves on IDP's performance and scalability is Skyline Dynamic Programming (SDP) [22].

When queries are optimized at the time they are submitted by the user, the selection process can add a substantial overhead to the execution time of the query. In order to avoid this, the Parametric Query Optimization (PQO) method was proposed. The goal here is to *apriori* identify the *parametric optimal set of plans* (POSP) for the entire parameter space at compile time, and subsequently to use at run time the actual parameter settings to identify the best plan – the expectation is that this would be much faster than optimizing the query from scratch. The PQO method was first proposed in [50] in the context of randomized algorithms for plan selection. They considered buffer size as the primary parameter, although their solution could work with arbitrary parameters. Subsequently, a number of PQO-based techniques have been proposed for cost-based optimizers:

In the pioneering work of Betawadkar & Ganguly [5], a System-R style optimizer with left-deep join-tree search space and linear cost models was built, the workload comprising of pure SPJ query templates with star or linear join-graphs and one-dimensional selectivity variations. They proposed the idea of finding an approximate POSP given a tolerance factor (cost increase threshold). Within this context, their experimental results indicate that, for a given cost increase threshold, plan reduction is more effective with increasing join-graph complexity. They also find that "*if the increase threshold is small, a significant percentage of the plans have to be retained*". For example, with a threshold of 10%, more than 50% of the plans usually have to be retained. However, this conclusion is possibly related to the low plan cardinality (less than 20 in all the experiments) in their original plan diagrams.

In subsequent work, Hulgeri & Sudarshan [47, 48] model an optimizer along the lines of the Volcano query engine [37], and evaluate SPJ query templates with two, three and four-dimensional relational selectivities. In their first paper [47], they discuss the PQO problem in the context of linear cost functions where the conventional optimizer is unaltered. The optimizer is treated as a black-box and the plans and costs returned

by the optimizer are used to find the POSP. They also propose a solution to the PQO problem in the presence of piecewise linear cost functions, which works for an arbitrary number of parameters. This solution involves altering the current optimizers to handle cost functions in the place of atomic cost values.

In the second paper [48], they propose a heuristic solution to the PQO problem which works with arbitrary nonlinear and discontinuous cost functions and any number of parameters. They propose an algorithm called *AniPQO* (and a variation of it called *DAG-AniPQO*), which requires minimal changes to existing optimizers and attempts to find a subset of the POSP such that for each point in the parameter space, either the optimal plan or a close-to-optimal plan is in the result set. The closeness to optimality is measured by an optimality threshold, which is guaranteed to be maintained in the case of linear cost functions, but cannot be guaranteed in the presence of nonlinear cost functions, when it is used only as a heuristic. Even with this relaxation, the final number of plans with a threshold of 10% can be large – for example, a 4-D query template with 134 original plans is reduced only to 53 with the DAG-AniPOSP algorithm and to 29 with AniPOSP.

Most of the solutions to the PQO problem are based on assuming cost functions that would result in one or more of the following:

1. Plan Convexity: If a plan P is optimal at point A and at point B, then it is optimal at all points on the line joining the two points;

2. Plan Uniqueness: An optimal plan P appears at only one contiguous region in the entire space;

3. Plan Homogeneity: An optimal plan P is optimal within the entire region enclosed by its plan boundaries.

However, it has been found that none of the three assumptions hold true, even approximately, in the plan diagrams produced by the commercial optimizers [64]. Even in situations where these assumptions hold, it is very difficult to store the regions of optimality of each of the plans in the POSP, so as to pick the best one at the time of execution. An alternative proposed by Hulgeri & Sudarshan in [47, 48], is to estimate the cost of all

the plans that belong to the POSP at the time of execution and pick the one that gives the minimum cost for the actual parameter values. This will be faster than optimizing the query from scratch, provided the number of plans in the POSP is not too large.

2.1.3 Run-time Refinements of Plan Choices

Query optimizers often make poor decisions because their compile-time cost models use inaccurate estimates of various parameters. There have been several efforts in the past to address this issue, which can be categorized as – strategies that make decisions at the start of query execution and strategies that make decisions during query execution. There are some parameters, like memory availability, whose value cannot be predicted at compile-time, but are accurately known at the start of execution. Assuming that the values of these parameters remain constant for the duration of the execution, the following strategies have been proposed:

1. Perform query optimization just before query execution. This method is not very efficient, especially if the query is executed repeatedly.

2. Find the best execution plan for all possible values of the parameters and lookup the best plan for the current parameter values at runtime (PQO).

3. Perform part of the optimization at compile time and defer any decisions that are affected by the parameter values to execution time.

For parameters whose value cannot be predicted at the start of the execution, like predicate selectivities, the following strategies can be applied:

1. Kabra and DeWitt [52] propose a technique which uses the query optimizer to generate a plan with each step of the plan containing the expected cost and result size statistics associated with it. At strategic points in the query execution, the intermediate runtime statistics are collected and compared to the the expected statistics and this information is used to alter the allocation of shared resources and if necessary, change the query execution plan itself. While changing the query execution plan,

they ensure that the operations already performed are not wasted, and re-optimize only the remainder of the query.

2. Antonshenkov [3] proposes a strategy where, in order to execute a query, multiple query plans are run in parallel. When one plan finishes or makes significant progress, the other competing plans are killed. This strategy assumes that ample resources are available, and is applied only to subcomponents of the query (typically to individual table accesses).

3. Roy et al [67] propose to optimize a batch of queries simultaneously so as to reuse common sub-expressions between the queries. They use an AND-OR DAG representation to compactly represent alternative query plans and extend this representation to ensure that all common sub-expressions are detected.

2.2 Random sampling

Random sampling techniques have been used extensively in various fields of DBMS when a quick synopsis regarding the underlying data is required. The most common use of sampling in query optimization is for query result size estimation, or for building the statistics. Following are some of the applications of random sampling techniques.

2.2.1 Approximate answer of aggregate queries

In current databases the amount of data poses a bottleneck for answering a query efficiently, as it involves scanning each and every tuple individually. Most often in decision support systems a less accurate answer is satisfactory to serve the purpose. The acceptability of inexact query answers coupled with the necessity for fast query response times has led researchers to investigate approximate query answering techniques that sacrifice accuracy to improve running time. This is known as approximate query processing (AQP). AQP is particularly common in data warehousing applications, which stores samples from the complete data set to decrease response time. Reader can refer to the

Aqua project from Bell Labs [2] for an example of this. The survey, "Sampling Methods In Approximate Query Answering Systems" composed by Gautam Das [21] illustrates some of the well known research work done on this field. The survey talks about the general rule in which most approximate query processing systems operate: first, during the "pre-processing phase", some auxiliary data structures, or data synopses, are built over the database; then, during the "runtime phase", queries are issued to the system and approximate query answers are quickly returned using the data synopses built during the pre-processing phase. The quality of an approximate query processing system is often determined by how accurately the synopsis represents the original data distribution. Instead of using the simple random sampling problem, researchers tried to incorporate the database specific knowledge for improving accuracy. Some of the procedures are described below:

- Ganti et al [32] proposed a heuristic pre-computation procedure called "Icicles", which biases the tuples according to how many times they have been accessed by different queries in the workload and assigns them with greater probabilities of being selected into the sample.

- Chaudhuri et al [10] proposed "Outlier Indexing" for improving sampling-based approximations for aggregate queries when the attribute being aggregated has a skewed distribution.

- Acharya et al [1] introduced "Congressional Sampling" which is specifically suited for answering group by queries with aggregation more efficiently.

- The stratified sampling approach was studied by Chaudhuri et al [8, 9], where authors formulated the problem of pre-computing a sample as an optimization problem, whose goal is to minimize the error for the given workload.

- Later Babcock et al [4] introduced the concept of dynamic sampling where the sampling technique attempts to strike a middle ground between pre-computed and online sampling.

2.2.2 Building and maintaining database statistics

There has been great deal of research on how to build the summary statistics efficiently using random sampling techniques. The ability to successfully use random sampling also requires that there should be a mechanism for determining the degree of sampling necessary for the desired accuracy. Several practical considerations make this problem challenging:

1. Need small error for all queries, with high probability.
2. Data distribution is not known a priori.
3. Columns have an unknown number of duplicate values.
4. For large samples, sampling at the tuple level is prohibitively expensive

Poosala et al [61] proposed elegant solutions for producing efficient histograms with random sampling techniques, which helps in keeping the cost of such production reasonably low. The main idea behind this approach is to perform the histogram construction looking at the sampled values instead of the entire relation.

Strategies on improving the tuple based random sampling to disk page or disk block level have been explored by Surajit Chaudhuri et al [11, 12].

2.2.3 Query optimization

Accurate and inexpensive estimation of database query sizes is useful for many purposes. Such estimates are used by query optimizers, to compare costs of alternative intermediate sub-plans. Sampling based query cardinality estimation procedures have received increasing attention over the past few years, as they not only give an estimate of the size of the query result, but also provide an indication of the precision of the estimate. Lipton and Naughton [55] and Lipton, Naughton and Schneider [56] presented adaptive sampling algorithms for estimating sizes of various *select* and *join* queries. However, the adaptive sampling algorithms proposed in [55, 56] assume knowledge of maximum size of each

sub-query. Since this is not usually available, an upper bound for this quantity is used in the expression for the termination criterion of the adaptive sampling approach, which can lead to taking considerably more observations than necessary causing the sampling algorithm to be unduly expensive in some cases. Later Haas and Swami [42] proposed improvements on the termination criterion. In the same paper [42] they also suggested classifying the tuples into different predefined groups (this requires knowledge about the data distribution) and then apply stratified sampling on each such group. Further refinements on the adaptive sampling algorithms were given by Haas et al [43], and again later by Haas et al [40].

In the work by Haas and Swami [42], the classification of tuples into groups is known a priori, which may not be feasible in practice. To overcome this, Ganguly et al [29] presented "Bifocal Sampling" which classifies the tuples into two categories namely sparse and dense, based on the number of tuples with the same join value, then applies distinct estimation strategy for these groups. The algorithm itself infers the class boundary at runtime while scanning the data.

There is an extensive survey done by Frank Olken [60] on efficient methods of answering random sampling queries of relational databases, motivated by the above issues. The author suggested introducing a random sample operator in the DBMS and laid out different algorithms to apply the operator in different situations.

2.3 Study of the Distinct Classes Estimators

The number of distinct values of an attribute in a relation is one of the critical statistics necessary for effective query optimization. It is well-established [39] that a bad estimate to the number of distinct values can slow down the query execution time by several orders of magnitude. Unfortunately, as the amount of data stored in a database increases, it becomes increasingly difficult to estimate the count of distinct values quickly with reasonable accuracy. Several approaches have been considered in the literature to deal with this issue. Recently, much of the work has focused on streaming models, or algorithms which are allowed to take only a single pass over the data. The challenge for these

algorithms lie in minimizing the space used, since the naive schemes run out of memory long before a single scan is complete. Another natural approach is to take a small random sample from the large dataset and then to estimate the number of distinct values from the sample. We now talk about some of the estimators devised following the second approach.

The initial work on distinct class estimators was presented by Goodman [35]. The Goodman's estimator works as follows: first, select m tuples at random from R, where R has n tuples. Next, delete unwanted attributes from the sampled tuples, and group the resulting tuples by the number of duplicates. That is, group i contains all tuples that, after deleting the projected-out attributes, appear i times in the sample. Let the number of tuples in group i be x_i. The Goodman's estimator estimates that the project has $\sum_{i>0} A_i x_i$ tuples, where,

$$A_i = 1 - (-1)^i \frac{\binom{n-m+i-1}{i}}{\binom{m}{i}}$$

Hou et al [45, 46] proposed Goodman's Estimator for use in the database setting for finding out distinct values of an attribute. The advantage of this estimator is that it requires no additional information regarding data distribution, hence with a moderate sample size it outperformed many well known estimator available. But the apparent drawbacks were that it is quite unstable with small sample sizes and causes huge computational overheads for large relations.

Haas et al [39] published a comprehensive review on different kinds of estimators for counting distinct values of an attribute. They showed empirically that the relative performance of the estimators is sensitive to the degree of skew in the data and finally proposed a new estimator that is a hybrid of the smoothed jackknife estimator (developed by them for low skew) and an estimator due to Shlosser [70](for high skew).

The above estimators are mostly based on heuristics and are not at all supported by any probabilistic error guarantees. An explanation for the apparent difficulty of distinct-values estimation was provided in the powerful negative result of Charikar et al [7]. They demonstrate two data distribution scenarios where the numbers of distinct values differ dramatically, yet a large number of random samples is required to distinguish between

the two scenarios. For example, to guarantee that an estimate has less than 10% error with high probability, requires sampling almost the entire table. They developed GEE (Guaranteed Error Estimator) which is armed with an expected error guarantee. We will discuss more about GEE in Section 3.2.

2.4 Behavior of Industrial Strength Optimizers

We now shift focus to studying the behavior of industrial strength query optimizers in practice. In [73], Waas and Galindo devise algorithms for counting, exhaustive generation, and uniform sampling of plans from the complete search space. Using this information, they study the cost distribution of query plans. Cost distributions are of interest because they can be taken as obvious indicators of the stochastic difficulty of a problem, by simply considering the ratio of high quality to low quality plans. They find that, under the precondition that the queries are of sufficiently large size, i.e., involving more than 4 or 5 joins, the distributions obtained "*correspond to Gamma-distributions with shape parameter close to 1*". They also find that the percentage of plans that are within *twice* the optimum cost is usually around 1% of the total number of plans in the search space.

Reddy and Haritsa [64] study the behavior of industrial strength optimizers from the perspective of the optimality space, instead of the search space. They examine the variation of the plan choices across the selectivity space and find that current optimizers make extremely fine-grained plan choices. They also observe that the plan optimality regions may have highly intricate patterns and irregular boundaries, indicating strongly non-linear cost models, that non-monotonic cost behavior exists where increasing result cardinalities decrease the estimated cost and, that the basic assumptions underlying the research literature on parametric query optimization often do not hold in practice. Further, there is heavy skew in the relative coverage of the plans, with 80 percent of the space typically covered by 20 percent or less of the plans. They show that through a process of plan reduction where the query points associated with a small-sized plan are swallowed by a larger plan, it is possible to bring down the cardinality of the plan diagram to about *one-third* of the original cardinality, without materially affecting the

query cost.

In this thesis, we study the problem of approximating *Optimizer Diagrams* i.e. plan diagrams (which represent the parametric optimal set of plans over the selectivity space), cost and cardinality diagrams arising from industrial-benchmark-based query templates operating on commercial state-of-the-art query optimizers. Our results indicate that it is indeed possible to generate 90% accurate diagram with only 15% overhead compared to the standard brute force approach.

Chapter 3

Plan Diagram Approximation

In this chapter, we describe our suite of strategies for the efficient generation of approximate plan diagrams. We begin with algorithms for Class I optimizers, and then describe how these techniques can be improved for Class II optimizers leveraging their foreign-plan-costing (FPC) feature. We conclude with two variants of an algorithm for Class III optimizers with FPC and plan-rank-list (PRL) functionalities – the first version *guarantees zero error*, while the second trades error for further reduction in computational overheads.

For ease of presentation, we will assume in the following discussion that the query template is 2D – the extension to d-dimensions is straightforward and given in appendix. The true plan diagram is denoted by P and the approximation as A, with the total number of query points in the diagrams denoted by n. Each query point is denoted by $q(x, y)$, corresponding to a unique query with selectivities x, y in the X and Y dimensions, respectively. The terms $p_P(q)$ and $p_A(q)$ are used to refer to the plans assigned to query point q in the P and A plan diagrams, respectively (when the context is clear, we drop the diagram subscript). We denote the total number of plans present in true diagram as α and in approximate diagram obtained after s explicit optimizations as α_s.

Finally, the plan identity and plan location errors of an approximate diagram are defined as,

$$\epsilon_I = \frac{\alpha - \alpha_s}{\alpha} * 100$$
$$\epsilon_L = \frac{|p_A(q) \neq p_P(q)|}{n} * 100$$

The approximation techniques should ideally ensure that ϵ_I and ϵ_L are within the user-specified bounds θ_I and θ_L, or are at least in their close proximity. In the process of approximation we estimate the value of ϵ_I and ϵ_L, we will use $\hat{\epsilon}_I$ and $\hat{\epsilon}_L$ to denote the estimated values (absolute) of ϵ_I and ϵ_L respectively.

3.1 Notation

Before we move ahead with presenting algorithms developed for plan diagram approximation we want reader to be familiar with the notations used in this chapter and later in the thesis. The following table consolidates the notations and abbreviations used:

Notations	
r	Resolution of the selectivity space
d	Dimension of the selectivity space
n	# of query points in the selectivity space (population size)
s	# optimizations performed (# samples)
P	True plan diagram
A	Approximate plan diagram
α	# of plans in the true diagram
α_s	# plans in approximate diagram after s optimizations
ϵ_I	True identity error
ϵ_L	True location error
$\hat{\epsilon}_I$	Estimated identity error
$\hat{\epsilon}_L$	Estimated location error
θ_I	User given identity error

θ_L	User given location error
σ	Risk factor of wrongly assigning an inferred point
x, y	Selectivity variable in 2D
q_o	An optimized point
q_u	An un-optimized point
q_{in}	An inferred point
p_k	An arbitrary plan
a_k	Area covered by plan p_k

PCM	Plan Cost Monotonicity
FPC	Foreign Plan Costing
PRL	Plan Rank List
PQO	Parametric Query Optimization
LPF	Low pass filter
QTx	Query Template $< x >$ of TPC-H benchmark
DSQTx	Query Template $< x >$ of TPC-DS benchmark
RS_NN	Random Sampling with Nearest Neighbor interpolation
GS_PQO	Grid Sampling with PQO assumption

3.2 Class I Optimizers

The approximation procedures for this class of optimizers operate in two main steps:

Optimization: A set of query points in the plan diagram are explicitly optimized to obtain the optimal plans at those points.

Inference: The plans for a set of un-optimized points are *inferred* using the information obtained from the *Optimization* step.

These steps are executed in an interleaved fashion under the following two main estimation phases that form the core of the sampling procedures described later in this

chapter.

ϵ_I **estimation:** In this phase, optimizations are performed to bring down ϵ_I below user given threshold θ_I.

ϵ_L **estimation:** In this phase we intend to achieve $\epsilon_L < \theta_L$ by performing an iterative process of optimization and inference.

Since we have empirically found that the plan-location error ϵ_L almost always lags plan-identity error ϵ_I, the performance of the algorithms were found to be more effective when ϵ_I *estimation* phase precedes ϵ_L.

3.2.1 Random Sampling with NN Inference (RS_NN)

In the RS_NN algorithm, we first use the classical random sampling technique to sample query points from the plan diagram that are to be optimized during the optimization step. Termination of RS_NN algorithm is based on both ϵ_I and ϵ_L estimators. In first pass we optimize a certain amount of query points based on ϵ_I estimator and infer the remaining un-optimized points. After that we evaluate ϵ_L estimator and if it meets the desired goal i.e. $\hat{\epsilon}_L \leq \theta_L$, we stop the algorithm there otherwise we optimize some more points and recheck the value of $\hat{\epsilon}_L$. This process iterates until the goal is properly met.

We now describe the two estimation phases devised to aid the RS_NN algorithm.

3.2.1.1 ϵ_I estimation phase

The problem of finding the distinct plans in the plan diagram can be related to the classical statistical problem of finding distinct classes in a population [41] – details are mentioned in Section 2.3. Applying the recent results of [7], we obtain the following: Let s samples be taken from the plan diagram, α_s be the number of distinct plans in these samples, and let f_1 denote the number of plans occurring only *once* in the samples. Then, it is highly likely that the number of distinct plans α in the entire plan diagram is in the range $[\alpha_s, \hat{\alpha}_{max}]$. $\hat{\alpha}_{max}$ is defined as,

$$\hat{\alpha}_{max} = (\frac{n}{s} - 1)f_1 + \alpha_s \tag{3.2.1}$$

If we ensure that the sampling is iteratively continued until α_s is within θ_I of $\hat{\alpha}_{max}$, then it is highly likely that the number of plans found thus far in the sample is within θ_I of α. Therefore, the RS_NN algorithm continuously evaluates Equation 3.2.1 to determine when the optimization step can be terminated.

Our experience, as borne out by the experimental results has been that the above stopping condition may be too conservative in that it takes many more samples than is strictly necessary. Therefore to refine the stopping condition, we also studied the other estimator for α, $\hat{\alpha}_{GEE}$ (Guaranteed Error Estimator) defined in [7] as,

$$\hat{\alpha}_{GEE} = (\sqrt{\frac{n}{s}} - 1)f_1 + \alpha_s \tag{3.2.2}$$

which has an expected ratio error bound of $O(\sqrt{\frac{n}{s}})$. Later, rigorous experimental investigation suggested that this estimator seems to be lacking stability for some of the less dense diagrams. Table 3.3 in the Experimental Section, shows the performance of different estimators when used as a terminating condition for the RS_NN algorithm.

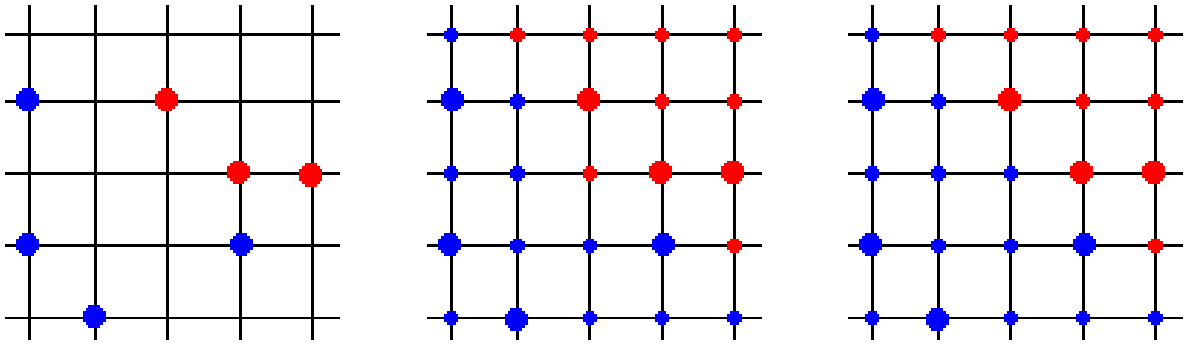
Therefore, we now terminate the optimization phase in two steps as follows: After α_s increases to a value within a $(1-\delta)$ factor of $\hat{\alpha}_{max}$, we continue the sampling until α_s reaches to within a $(1-\theta_I)$ factor of $\hat{\alpha}_{GEE}$. The value of δ conducive to good performance results has been empirically determined to be 0.3. The estimated value of ϵ_I i.e $\hat{\epsilon}_I$ is calculated as,

$$\hat{\epsilon}_I = \begin{cases} \dfrac{\hat{\alpha}_{max} - \alpha_s}{\hat{\alpha}_{max}} & \alpha_s \leq (1-\delta)\hat{\alpha}_{max} \\ \dfrac{\hat{\alpha}_{GEE} - \alpha_s}{\hat{\alpha}_{GEE}} & otherwise \end{cases} \tag{3.2.3}$$

The intuition behind this method is that once the gap between α_s and $\hat{\alpha}_{max}$ has narrowed to a sufficiently small range, then the $\hat{\alpha}_{GEE}$ estimator can be used as a reliable indicator of the plan cardinality in the diagram.

3.2.1.2 Inference

After the completion of the sampling phase, the plan choices at the un-optimized points of the plan diagram need to be inferred from the plan choices made at the sampled points.



(a) **Optimize** (b) **Infer** (c) **Filter**

Figure 3.1: Execution Stages of the RS_NN Algorithm

This is done using a Nearest Neighbor (NN) style classification scheme [72]. Specifically, for each un-optimized point q_u, we search for the nearest optimized point q_o, and assign the plan of q_o to q_u. If there are multiple nearest optimized points, then the plan that occurs most often in this set is chosen, and in case of a tie on this metric, a random selection is made from the contenders. This NN classification metric is based on *Naive Bayes Classifier* [27].

According to Bayes theorem [27], the finite risk ($r(x|c_i)$) of misclassifying x by class $|c_i|$ is defined as $1 - \Pr(c_i|x) = 1 - \dfrac{\Pr(x|c_i)\Pr(c_i)}{\Pr(x)}$. Now $\Pr(x|c_i) = \dfrac{|c_i|_{vicinity}}{|c_i|}$ and $\Pr(c_i) = \dfrac{|c_i|}{s}$. Considering $\Pr(x) = 1$ we can evaluate $r(x|c_i)$ as below,

$$r(x|c_i) = 1 - \frac{|c_i|_{vicinity}}{s} \tag{3.2.4}$$

To infer x, we choose c_i for which the risk in minimum i.e. we follow a *"majority wins"* policy for inference.

The distance between two query points $q_1(x_1, y_1)$ and $q_2(x_2, y_2)$ can be calculated using various distance metrics. We have evaluated the following three popular metrics: (1) *Manhattan* (L_1 norm); (2) *Euclidean* (L_2 norm); and (3) *Chessboard* (L_∞ norm). Our experience has been that the Chessboard Distance is most suitable, since the transition boundaries between plans often tend to be aligned along the (horizontal and vertical) axes. The same metric is also used for establishing the geometries of plan clusters in PLASTIC [33, 65], a tool designed to amortize query optimizer overheads.

3.2.1.3 ϵ_L estimation phase

We perform the ϵ_L estimation phase, after the ϵ_I estimation phase is over i.e. $\hat{\epsilon}_I \leq \theta_I$ and all the un-optimized points are inferred. This estimator is intended to capture errors performed by the NN inference technique.

Consider an inferred point q_{in}, whose correct plan is $p_P(q_{in})$. Now the NN interpolation technique could mistakenly assign an incorrect plan to q_{in} due to one of the following reasons,

Case 1: When the right plan p_k is not present in the approximate diagram. Let us find out probability of such an event. Suppose the area occupied by plan p_k is a_k, then with s samples taken with replacement from population of size n, probability of missing p_k is given $(1 - \frac{a_k}{n})^s$. It is evident that with high value of a_k, probability of missing p_k will decrease.

Now we discuss the effect of such an event on ϵ_L. For the plan p_k, ϵ_L may also increase by a value $\frac{a_k}{n}$ with a probability $(1 - \frac{a_k}{n})^s$. Hence, the expected increase in ϵ_L due to plan identity error is $\Delta(\epsilon_L) = \sum_{k=1}^{\theta_I \times \alpha} \frac{a_k}{n}(1-\frac{a_k}{n})^s$. Now the term $\frac{a_k}{n}(1-\frac{a_k}{n})^s$ attains maxima when $\frac{a_k}{n} = \frac{1}{s+1}$.

$$\begin{aligned}
\Delta(\epsilon_L) &= \sum_{k=1}^{\theta_I \times \alpha} \frac{a_k}{n}(1 - \frac{a_k}{n})^s \\
&< \sum_{k=1}^{\theta_I \times \alpha} \frac{1}{s+1}(1 - \frac{1}{s+1})^s \\
&< \theta_I \times \alpha \times \frac{1}{s+1}(1 - \frac{1}{s+1})^s
\end{aligned}$$

For $\theta_I = 10\%$ applying *bisection method* we found that by setting $s = 500$, we can achieve $\Delta(\epsilon_L) < 0.01$, if number of plans are restricted to couple of hundreds. Further, even if the number of plans are as high as 500, we can achieve the same performance with $s = 1500$. Our experience suggests that, the number of plans found in plan diagrams are around few hundreds, and we almost always optimize in excess of 500 query points. Therefore, the location error contribution due to Case 1 in our case is within an acceptable range.

Case 2: The right plan $p_P(q_{in})$ is present in the approximate diagram, but we still miss it due to the following reasons.

Case 2.1: $p_P(q_{in})$ is not part of the set of optimized points found at the nearest neighborhood. This can happen when the plan $p_P(q_{in})$ is scattered across the plan diagram. Generally speckle or very thin Venetian blind patterns[64] found in plan diagrams, may give rise to an error of this kind. Probabilistic computation is hard for this kind of error and outside scope of this thesis.

Case 2.2: $p_P(q_{in})$ is present in minority in the neighborhood i.e either q_{in} is victim of tie or surrounded by many plans.

We will concentrate on estimating error incurred due to the Case 2.2 in this thesis. According to our inference metric we designed the location error heuristic as,

$$\begin{aligned} \hat{\epsilon}_L &= \sum_{i=1}^{n} r'(q_i|p_A(q_i)) \\ &= \sum_{i=1}^{n} \left(1 - \frac{s_i(p_A(q_i))}{s_i}\right) \end{aligned} \tag{3.2.5}$$

where s_i is number of samples found at the nearest neighbor and $p_A(q_i)$ is the plan inferred at the point q_i in the approximate diagram A. The term $s_i(p_A(q_i))$ denotes the number of optimized points covered by $p_A(q_i)$ at the nearest neighbor. $r'(q_i|p_A(q_i))$ is a heuristic based on $r(q_i|p_A(q_i))$ given in Equation 3.2.4, where we replace s with s_i. We found this heuristic to be a better estimator for location error.

This estimator works as follows. We complete one iteration of optimization phase that brings down the value of $\hat{\epsilon}_I$ below θ_I followed by one iteration of Inference step. Once these are over we calculate the value of $\hat{\epsilon}_L$ using the Equation 3.2.5. If $\hat{\epsilon}_L$ is found to be greater than θ_L, we optimize few (1%) more query points. Based on the updated set of optimized points we then re-run the inference step and re-evaluate the estimator. This process is executed in an iterative fashion as long as $\hat{\epsilon_L} > \theta_L$.

Algorithm 1 The RS_NN Algorithm

RS_NN (QueryTemplate QT, IdnErrBound θ_I,LocErrBound θ_L, IncrSamples s)

**********Phase: ϵ_I Estimation (Step: Optimization)**********\
stage $\leftarrow$ 1;
while true do
  Optimize s samples chosen uniformly at random;{ * $s \geq 106$ *\}
  Compute the values of $\hat{\alpha}_{max}$ and $\hat{\alpha}_{GEE}$;
  if stage = 1 **then**
    if $\alpha_s \geq ((1-\delta)\hat{\alpha}_{max})$ **then**
      stage $\leftarrow$ 2;
    end if
  else if stage = 2 **then**
    if $\alpha_s \geq ((1-\theta_I)\hat{\alpha}_{GEE})$ **then**
      break;
    end if
  end if
end while
**********Phase: ϵ_I Estimation (Step: Interpolate)**********\
for each un-optimized point q_u **do**
  Determine the set m of nearest optimized neighbors;
  Determine plan p_k which occurs most often in m;
  In case of a tie set p_k by picking any plan at random from the contenders;
  Assign the plan p_k to q_u;
end for
**********Phase: ϵ_L Estimation**********\
while true do
  Calculate $\hat{\epsilon}_L = \sum_{j=1}^{n} \Pr[q_j$ has wrong plan $p_A(q_j)]$;
  if $\hat{\epsilon}_L \leq \theta_L$ **then**
    break;
  else
    Optimize s samples chosen uniformly at random;
    Apply interpolation step;
  end if
end while
**********Low Pass Filter**********\
for each inferred point q_{in} **do**
  Check for plan multiplicity at chessboard distance 1;
  if a plan p_k occupies more than half of the neighbors **then**
    Assign p_k to q_{in};
  end if
end for
return

3.2.1.4 Low Pass Filter (LPF).

Inference using the NN scheme is well-known to result in boundary errors [72] – in our case, along the plan optimality boundaries. To reduce the impact of this problem which is prominent with small sample size and large ϵ_L, we apply a low-pass filter [34] after the

final inference i.e. after completion of ϵ_L estimation phase. The filter operates as follows: For each inferred point q_{in}, all its neighbors (both optimized and inferred) at a distance of one, are examined to find the plan that is assigned to more than half of the neighbors. If such a plan exists, it is assigned to q_{in}, otherwise the original assignment is retained.

Note that each un-optimized point is processed once during the filter phase, and that the resultant diagram is dependent on the order in which the points are taken up for processing. Further, the filter could, in principle, be applied multiple times. However, our empirical results indicate that the choice of processing order has only minuscule impact on overall diagram accuracy – in our implementation, the points are processed starting from the top right corner and moving towards the origin in reverse row-major order. Also, applying the filter multiple times does not provide any perceptible improvement – therefore, we apply it only once.

The functioning of the RS_NN algorithm is illustrated in Figure 3.1 – in this set of pictures, each large dot indicates an optimized query point, whereas each small dot indicates an inferred query point. The initial set of optimized sample query points is shown in Figure 3.1(a), and the NN-based inference for the remaining points is shown in Figure 3.1(b). Applying the LPF filter results in Figure 3.1(c) – note that the center query point, which has an (inferred) red plan in Figure 3.1(b), is re-assigned to the blue plan in Figure 3.1(c).

The complete RS_NN algorithm is shown in Algorithm 1. In our implementation, the initial number of samples s_0 is set to 1% of the space, and the increment in the number of samples after each iteration is also set to this value.

3.2.2 Grid Sampling with PQO Inference (GS_PQO)

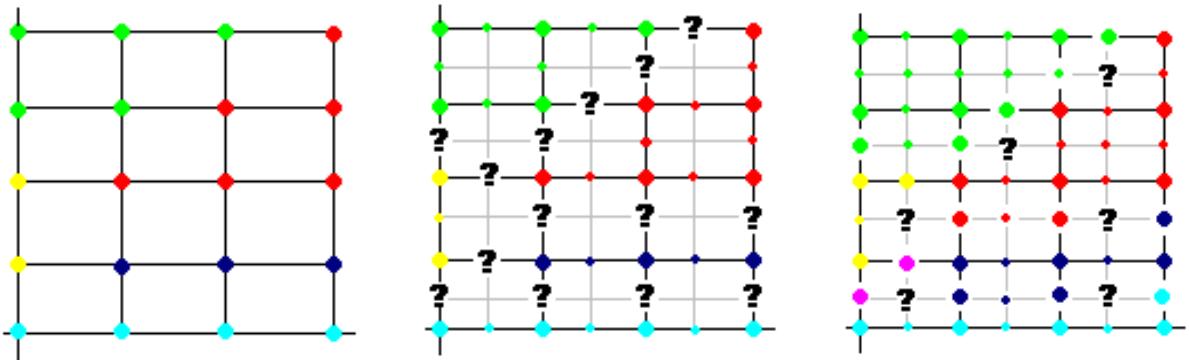
We now turn our attention to an alternative approach based on *grid sampling*. Here, a low resolution grid of the plan diagram is first formed, which partitions the selectivity space into a set of smaller rectangles. The query points corresponding to the corners of all these rectangles are optimized first. Subsequently, these points are used as the seeds to determine which of the other points in the diagram are to be optimized.

Specifically, if the plans assigned to the two corners of an edge of a rectangle are the same, then the mid-point along that edge is also assigned the same plan. This is essentially a specific inference based on the guiding principle of the Parametric Query Optimization (PQO) literature (e.g. [47]): "If a pair of points in the selectivity space have the same optimal plan p_k, then all locations along the straight line joining these two points will also have p_k as their optimal plan." At first glance, our usage of the PQO principle here may seem to be at odds with our earlier observation in [64] that, for industrial-strength optimizers, this principle is observed more in breach than in compliance. However, the difference is that we are applying PQO at a "micro-level", that is, within the confines of a small rectangle in the selectivity space, whereas earlier work has effectively considered PQO as a universal truth that holds across the entire space. Our experimental experience has been that micro-PQO generally holds in all the plan diagrams that we have analyzed.

When the plans assigned to the end points of an edge are different, then the midpoint of this edge is optimized. After all sides of a given rectangle are processed, its center-point is then processed by considering the plans lying along the "cross-hair" lines connecting the center-point to the mid-points of the four sides of the rectangle. If the two end-points on one of the cross-hairs match, then the center-point is assigned the same plan (if both cross-hairs have matching end-points, then one of the plans is chosen randomly). If neither of the cross-hairs has matching endpoints, the center-point is optimized. Now, using the cross-hairs, the rectangle is divided into four smaller rectangles, and the process recursively continues, until all points in the plan diagram have been assigned plans.

The progress of the GS_PQO algorithm is illustrated in Figure 3.2 (again, each large dot indicates an optimized point, whereas each small dot indicates an inferred point). Figure 3.2(a) shows the state after the initial grid sampling is completed. Then, the '?' symbols in Figure 3.2(b) denote the set of points that are to be optimized in the following iteration as we process the sides of the rectangles. Finally, Figure 3.2(c) enumerates the set of points that are to be optimized while processing the cross-hairs.

A limitation of the GS_PQO algorithm is that it may perform a substantial number of unnecessary optimizations, especially when a rectangle with different plans at its

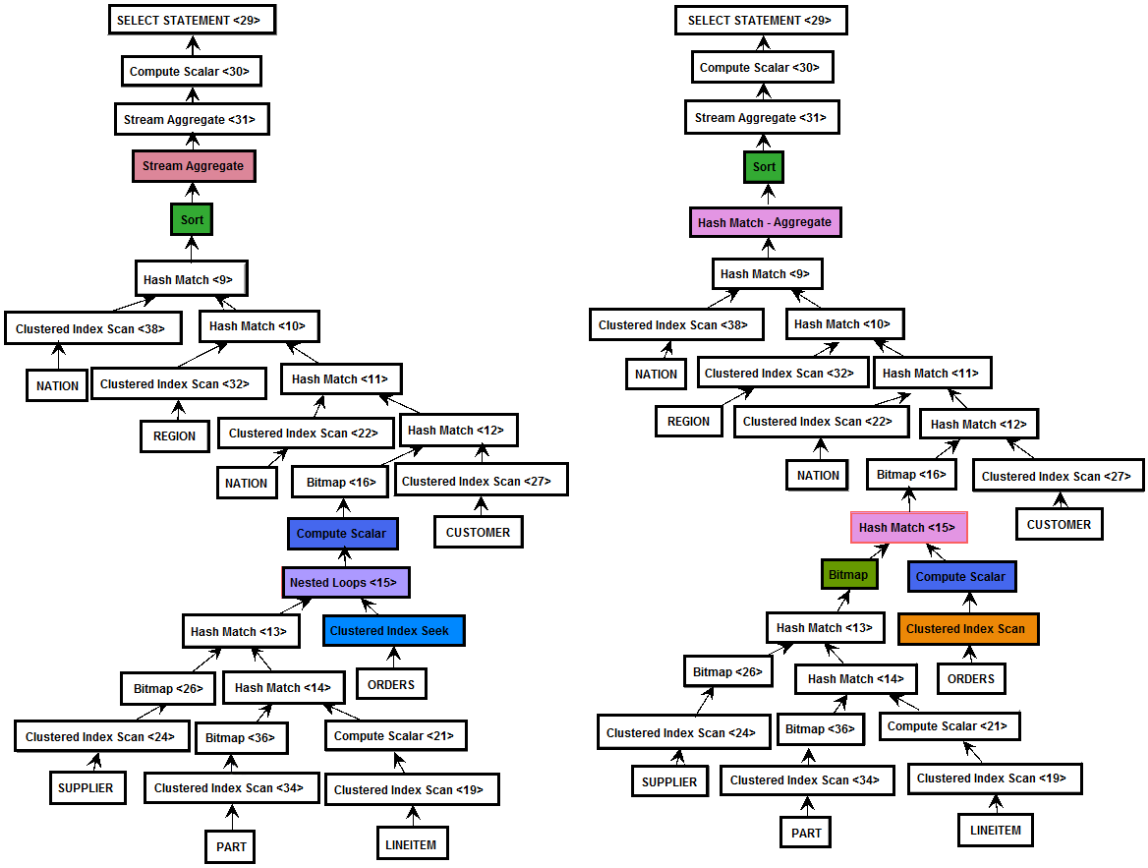


(a) **Initial Grid** (b) **Mid Points** (c) **Cross Hairs**

Figure 3.2: Execution Loop in GS_PQO Algorithm

endpoints features only a small number of new plans within its enclosed region. This is because GS_PQO does not distinguish between sparse and dense low-resolution rectangles. For example, if two similar-sized rectangles each have two plans featured at their four corner points, then they are divided similarly irrespective of the expected number of new plans present in the interior. We attempt to address this issue by refining the algorithm in the following manner: Assign each rectangle R with a "plan-richness" indicator $\rho(R)$ that serves to characterize the expected plan density in R, and then preferentially assign optimizations to the rectangles with higher ρ.

Our strategy to assign ρ values is as follows: Instead of merely making a *boolean* comparison at the corners of the rectangle as to whether the plans at these points are identical or not, we now dig deeper and compare the *plan operator trees* associated with these plans in order to estimate interior plan density. As an extreme example, consider the case where there is a left-deep tree at one corner of the rectangle, and a right-deep tree at another corner. In this situation, it seems reasonable to expect that there will be a significant number of plans in the interior of the rectangle since the process of shifting from a left-deep to a right-deep tree usually occurs in incremental intermediate steps, each corresponding to a new plan, rather than all at once – we have confirmed this observation through detailed analysis of the plan diagrams of industrial optimizers.

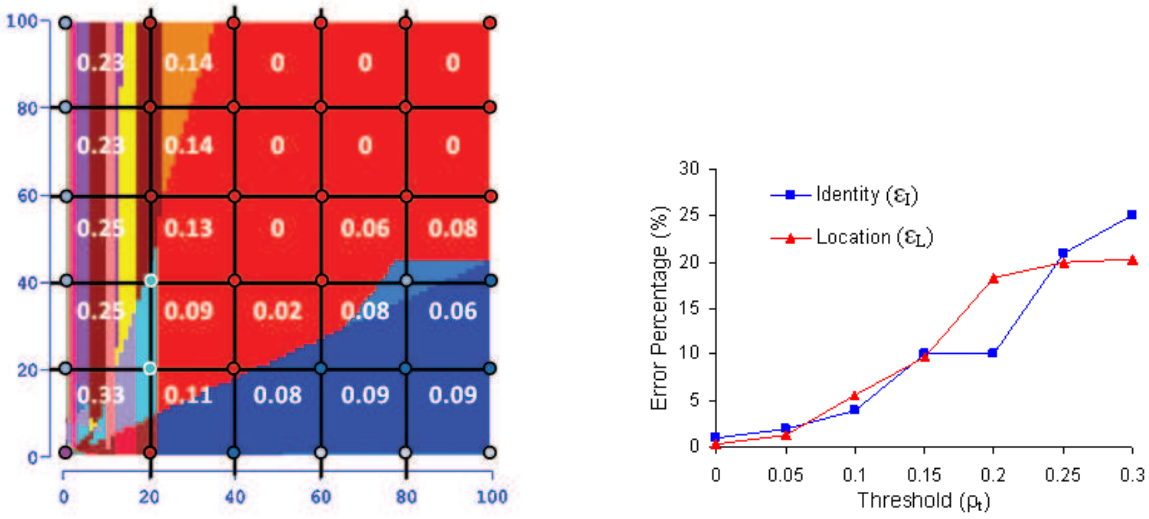


(a) Plan Tree T_1 (b) Plan Tree T_7

Figure 3.3: Example of Plan Tree Difference (QT8)

Plan Tree Differencing. Let the operator trees corresponding to a pair of plans p_i and p_j be denoted by T_i and T_j, respectively. Our comparison strategy is based on identifying and mapping similar operator nodes in the two trees. Figure 3.3 shows an example pair of plan trees T_1 and T_7 corresponding to the plans p_1 and p_7 that feature in the plan diagram of Figure 1.4(a) – the white nodes depict matching nodes, whereas the colored nodes represent distinct nodes.

In the following description, the term *branch* is used to refer to any directed chain of unary-input nodes between leaf and a binary node, or between a pair of binary-input nodes, in these trees. Branches are directed from the lower node to the higher node, modeling the direction of data propagation. The matching proceeds as follows:



(a) ρ values in Plan Diagram (QT9)

(b) Effect of ρ_t on ϵ_I and ϵ_L Errors

Figure 3.4: Behavior of ρ

1. First, all the leaf nodes (relations) and all the binary-input nodes (typically join nodes) are identified for T_i and T_j.

2. A leaf of T_i is matched with a leaf of T_j if and only if they both have the same relation name. In the situation that there are multiple matches available (that is, if the same relation name appears in multiple leaves), an edit-distance computation is made between the branches of all pairs of matching leaves between T_i and T_j. The assignments are then made in increasing order of edit-distances. For example, the NATION node appears twice in T_1 and T_7 of Figure 3.3, and the specific pairing is made based on the closeness of matching in the branches arising out of these nodes.

3. A binary node of T_i is matched with a binary node of T_j if the set of base relations that are processed is the same. If the node operator names and the left and right inputs are identical (in terms of base relations), the nodes are made white. However, if the node operator names are different, or if the left and right input relation subsets are different, then the nodes are colored.

4. A minimal edit-distance computation is made between the branches arising out of

each pair of matched nodes, and the nodes that have to be added or deleted, if any, in order to make the branches identical, are colored. Unmodified nodes, on the other hand, are matched with their counterparts in the sibling tree and made white.

5. Finally, each pair of matched nodes is assigned the same unique number in both trees. For example, the number 15 is assigned to the final join node, representing the composite relation formed by the join of all the base relations, in each tree.

Plan Richness Metric. We now describe the procedure to quantify plan richness in terms of plan-tree differences. Our formulation uses $|T_i|$ and $|T_j|$ to represent the number of nodes in plan-trees T_i and T_j, respectively, and $|T_i \cap T_j|$ to denote the number of matching nodes between the trees.

Now, ρ is measured as the classical Jaccard Distance [72] between the trees of the two plans, and is computed as

$$\rho(T_i, T_j) = 1 - \frac{|T_i \cap T_j|}{|T_i \cup T_j|} \tag{3.2.6}$$

For example, the ρ for the plan tree pair (T_1, T_7) in Figure 3.3 is $1 - \dfrac{28}{28+5+6} = 1 - \dfrac{28}{39} = 0.28$.

While Equation 3.2.6 works for a pair of plans, we need to be able to extend the metric to handle an arbitrary set of plans, corresponding to the corners of the hyper-rectangle in the selectivity space. Given a set of n trees $\{T_1, T_2, \ldots, T_n\}$, this is achieved through the following computation:

$$\rho(T_1, \ldots, T_n) = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \rho(T_i, T_j)}{\binom{n}{2}} \tag{3.2.7}$$

Note that the ρ values are normalized between 0 and 1, with values close to 0 indicating that all the plans are structurally very similar to each other, and values close to 1 indicating that the plans are extremely dissimilar. Figure 3.4(a) depicts the ρ values calculated for a sample plan diagram (produced from a query template based on TPC-H Query 9) after partitioning into 20x20 squares. We see here that ρ reaches high values

close to the origin and along the selectivity axes. This is in accordance with earlier observations in [47, 48, 62, 63, 64] that plans tend to be densely packed in precisely these regions of the selectivity space.

3.2.2.1 ϵ_I estimation phase

We now describe how GS_PQO utilizes the above characterization of plan-tree-differences. First, the grid sampling procedure is executed as mentioned earlier. Then, for each resulting rectangle, the ρ value is computed based on the plan-trees at the four corners, using Equation 3.2.7. The rectangles are organized in a *max-Heap* structure based on the ρ values, and the optimizations are directed towards the rectangle R_{top} at the top of the heap, i.e. with the current highest value of ρ. Specifically, the PQO principle is applied to the mid-points of all qualifying edges (those with common plans at both ends of the edge) in R_{top}, and all the remaining edge mid-points are explicitly optimized. The rectangle is then split into four smaller rectangles, for whom the ρ values are recomputed, and these new rectangles are then inserted into the heap. This process continues until all the rectangles in the heap have a ρ value that is below a threshold ρ_t. Next we discuss how we set the value of the threshold ρ_t.

Setting value of ρ_t. A representative behavior of the error metrics, ϵ_I and ϵ_L, as a function of the ρ_t threshold, is shown in Figure 3.4(b), obtained with query template QT8. Note that we find here that ϵ_L does not always lag ϵ_I. The reason is that samples cease to be assigned to rectangles with $\rho \leq \rho_t$ even when they contain more than one plan. In this situation, our current inference scheme may increase ϵ_L due to erroneous boundary detection between the plans present inside the rectangle. Therefore, even when ϵ_I is low, ϵ_L may be comparatively large. We see in Figure 3.4(b) that setting the threshold equal to the error bound(θ_I), i.e. $\rho_t = \theta_I$ (e.g. for $\theta_I = 10\%, \rho_t = 0.1$), is an adequate heuristic that is sufficient to meet user expectations on both error metrics – we observed similar behavior for most of the other query templates as well, and therefore applied this heuristic in [24]. Further investigation revealed that for some of the query templates ρ_t is too loose and for some it was not sufficient. In this thesis we propose another heuristic which is

seen to be working pretty well in almost all the cases. To do so, we identified five factors as mentioned below, that affects determining the value of ρ_t.

1. **Query point distribution :** This is taken care of implicitly by the initial hyper-rectangle size and position defined, e.g for exponential distribution the hyper-rectangles are implicitly of smaller size around origin and bigger elsewhere.

2. **Resolution of the plan diagram (r):** The initial grid size is taken to be $\sqrt{r}$, which is a sub-linear scale up. This ensures that we don't miss too many plans with increasing resolution. We have verified through experiments that the same ρ value in higher resolution may depict higher number of plans than its low resolution counterpart.

3. **Dimension of the plan diagram (d):** Same ρ value for same template differing in dimension may not indicate same amount of plan-richness, therefore we should design different ρ_t according to dimension of the query template. However, while calculating ρ value for a rectangle R, we consider the number of corner points, which already takes care of the dimension factor.

4. **Maximum ρ value (ρ_{max}) :** Similarly same ρ value for two different query templates quantify different plan density. Our experience has been that the maximum ρ value found while processing the initial rectangles can be used to quantify the expected number of plans at the corner points throughout the diagram.

5. **Frequency of plans inside the ρ_{max} rectangle (ξ):** The number of plans present at the corners of the highest ρ hyper-rectangle reflects the correlation between ρ and the incremental plan structure difference for a particular plan diagram. For example if ρ_{max} is achieved by k plans in a certain plan diagram then we say to introduce a new plan a gap of approximately $\frac{\rho}{k}$ value is required. Similarly in another case if the same ρ_{max} value is achieved by k' plans where $k' > k$, we would want to tighten ρ_t further. Therefore we find this factor important in deciding the value of ρ_t. Now assume there exists two different plan diagrams A_1 and A_2, both having $\rho_{max} = 0.4$

and k plans at the corners. But among the k corner plans in A_1 assume $k-1$ plans are structurally very similar whereas for A_2 all the k plans differ substantially. According to our strategy ρ_t will be set to the same value for both A_1 and A_2, which is not desirable. Therefore, instead of simply counting the number of plans at the corners of the ρ_{max} rectangle, we use a heuristic which evaluates it according to the structural differences of those plans.

We introduce "Contribution Factor"(ξ) heuristic as the quantitative measurement of the plan frequency at the corners of a hyper-rectangle R_k. We use ρ calculation to estimate ξ_k of a hyper-rectangle R_k with 2^d (d - dimension) corner points as,

$$\xi_k = \#\ plans \times \sum_{i=1}^{2^d} \min_{(j=1:2^d, i\neq j)} \rho(T_i, T_j) \tag{3.2.8}$$

To illustrate the calculation of ξ_k let us consider a rectangle with distinct plans at the four corners. Now suppose the set $X = \{\{0, 0.33, 0.25, 0.01\}, \{0.33, 0, 0.23, 0.3\}, \{0.25, 0.23, 0, 0.27\}, \{0.01, 0.3, 0.27, 0\}\}$ denotes the ρ calculated for each pair of plans. The contribution factor of such rectangle is calculated as, $\xi = 4 \times (0.01 + 0.23 + 0.27 + 0.01) = 2.08$. Note that according to our heuristic ξ is allowed to take a value $> 2^d$ in case there exists huge structural difference among the corner plans.

Among the five factors mentioned above we argued that query distribution and dimension of the selectivity space were taken care of implicitly by GS_PQO. Now we design our heuristic ϵ_I estimator taking into account the remaining three factors as follows,

1. initial grid size is set to $\lceil \sqrt{r} \rceil$
2. ρ_t is determined as: $\rho_t = \dfrac{\rho_{max}}{\xi_{max}} \times \theta_I$

This heuristics shows reasonable performance for both uniform and exponential query distribution which is evident from the experimental results presented in Section 3.5..

Final Inference. As mentioned earlier, GS_PQO uses the PQO-based inference technique within each rectangle until its plan richness metric goes below the ρ_t threshold. After the threshold is crossed, there may still be unassigned points within the rectangle.

These are handled as follows in the final inference phase: The same PQO-based inference scheme is used with the only difference being that whenever an edge has different end-points, then the plan assignment of the mid-point is done by randomly choosing one of the end-point plans, rather than resorting to explicit optimization.

Algorithm 2 The GS_PQO Algorithm

GS_PQO (QueryTemplate QT, IdentityErrorBound θ_I, LocationErrorBound θ_L)

**********Phase : ϵ_I Estimation (Step: Optimization)**********\
Optimize the points in the initial low-resolution grid with edge length set to $\sqrt{r}$;
Calculate ρ for each rectangle using Equation 3.2.7;
Organize the rectangles in a max-Heap (M_ρ) based on their ρ values;
$\rho_t \leftarrow \frac{\rho_{max}}{\xi_{max}} \times \theta_I$ (Class I optimizers) | $\rho_t \leftarrow \sqrt{d} \times (\frac{\rho_{max}}{\xi_{max}} \times \theta_I)$ (Class II optimizers);
for the rectangle R_{top} at the top of M_ρ **do**
  if $\rho(R_{top}) \leq \rho_t$ **then**
    break;
  else
    Extract R_{top} from M_ρ;
    Apply PQO inference to mid-points of qualifying edges of R_{top};
    Optimize all the remaining mid-points q_o. Set $\sigma(q_o) = 0$;
    Split R_{top} into four equal rectangles and compute ρ values for the smaller rectangles;
    Insert the new rectangles into M_ρ;
  end if
end for
**********Phase : ϵ_I Estimation (Step: Interpolation)**********\
while the heap is not empty **do**
  Extract R_{top} from the M_ρ;
  Add a copy of R_{top} into a max-Heap $M_{\hat{\epsilon}_L}$ ordered on its $\hat{\epsilon}_L$ contribution;
  {* *This heap will later be used for ϵ_L estimation phase* *\}
  Select a plan at random from the edge end points and assign it to the mid point q_u;
  $\sigma(p_A(q_j)) \leftarrow 1 - \frac{s(c_t)(1-\sigma(q_j))}{s}$ if assigned with plan at point q_j;
  Recursively split the rectangles until all points inside R_{top} are processed;
end while
**********Phase: ϵ_L Estimation**********\
while true do
  $\hat{\epsilon}_L \leftarrow \sum_{j=1}^{n} \sigma(q_j)$;
  if $\hat{\epsilon}_L \leq \theta_L$ **then**
    break;
  else
    Optimize s points extracting from $M_{\hat{\epsilon}_L}$;
    Insert the sub-rectangles in $M_{\hat{\epsilon}_L}$;
    Infer the un-optimized points again;
  end if
end while

3.2.2.2 ϵ_L estimation phase

It is evident that the stopping condition devised using $\rho(R)$ takes care of ϵ_I only. So similar to RS_NN we developed a location error heuristic for GS_PQO too. We use the term *risk factor (σ)* to denote the estimation of the probability of misclassifying an un-optimized point. Naturally the *risk factor* for an optimized point is 0. Suppose an inferred point q_{in} is surrounded by points $q_1, q_2, ..q_k$ in a d-dimensional plan diagram, whose risk factors are $\sigma(q_1), \sigma(q_2), \ldots \sigma(q_k)$ respectively. Now suppose q_{in} gets assigned from a neighboring point q_j associated with plan p_j, then the risk factor at q_{in} following is set to,

$$\sigma(q_{in}) = 1 - \frac{s_{in}(p_j)(1 - \sigma(q_j))}{s_{in}} \tag{3.2.9}$$

where s_{in} is the number of points (both optimized and inferred) in the neighborhood consulted for inferring q_{in} and $s_{in}(p_j)$ is the number of such points assigned with plan p_j. One might wonder that how can we calculate the value of $\sigma(q_1), \sigma(q_2), \ldots \sigma(q_k)$ beforehand. Note that the GS_PQO works by first optimizing points from the initial grid, each of whose σ values (which is 0) are known. This is used in second iteration for assigning σ value to the set of un-optimized points encountered, which in turn helps in assigning σ for un-optimized points discovered so far. The intuition behind calculating the additional multiplicative factor $s(p_j)$ is that in GS_PQO we do not always infer from an optimized point, therfore we included the extra term $(1 - \sigma(q_j))$. We estimate $\hat{\epsilon}_L$ in similar fashion as we did for RS_NN,

$$\hat{\epsilon}_L = \sum_{j=1}^{n} \sigma(q_j) \tag{3.2.10}$$

If the value of $\hat{\epsilon}_L$ is evaluated to be greater than θ_L then we follow the steps below to reduce the location error iteratively,

1. Add the rectangles with $0 < \rho < \rho_t$ earlier rejected for further optimizations in a max heap $M_{\hat{\epsilon}_L}$, this time ordered according to the amount of location error they are contributing.

2. Extract the rectangle with maximum $\hat{\epsilon}_L$ contribution from $M_{\hat{\epsilon}_L}$. The rectangle is divided into sub rectangles by optimizing the mid points and cross-hair.

3. Keep on dividing the rectangles till the extra optimizations are within 1% of n.

4. Infer plans for all the un-optimized points again and evaluate the $\hat{\epsilon}_L$ estimator, if it is still greater than θ_L we repeat the process from Step 2.

The complete GS_PQO algorithm is shown in Algorithm 2.

3.2.3 Approximation Overhead Estimator

While approximating a Optimizer Diagram appears sufficiently fast, it is likely that users might need to know a rough estimate of the approximation time before they start generating. For example, user would like to tune the error threshold to meet his/her purpose. Therefore, it is helpful to design a fast estimator to produce a rough estimate of the time required. Charikar et al [7] proved a strong negative result showing that any such estimation with low sample size cannot provide strong guarantee.

We developed approximation time estimators for both RS_NN and GS_PQO process applicable only for Class I and II optimizers as described below:

S-EST: The estimator developed for RS_NN is called S-EST. It starts by optimizing a small number of query points using simple random sampling technique. Then it estimates identity error applying a conservative statistical estimator α_{max} mentioned in Section 3.2.1.1. It further optimizes some more points and calculates the improvement in identity error estimate. The number of samples required for this improvement is linearly scaled up to the desired improvement in terms of user given error threshold. For example look at the Figure 3.5, where the *red* line shows the error estimates obtained by α_{max} and the *blue* line shows the actual error incurred by the approximation process with increasing sample size. S-EST works by extending the red line till it reaches the θ_I threshold in the plot, which is 10% in this case. The choice of initial sub-set of points was to improve the population to sample fraction used in the α_{max}, which tends to overestimate the actual number of plans if the population to sample ratio is too high.

Note that our estimation process only considers plan-identity error. This is because, given similar tolerance levels for both errors, our empirical observation has been that the

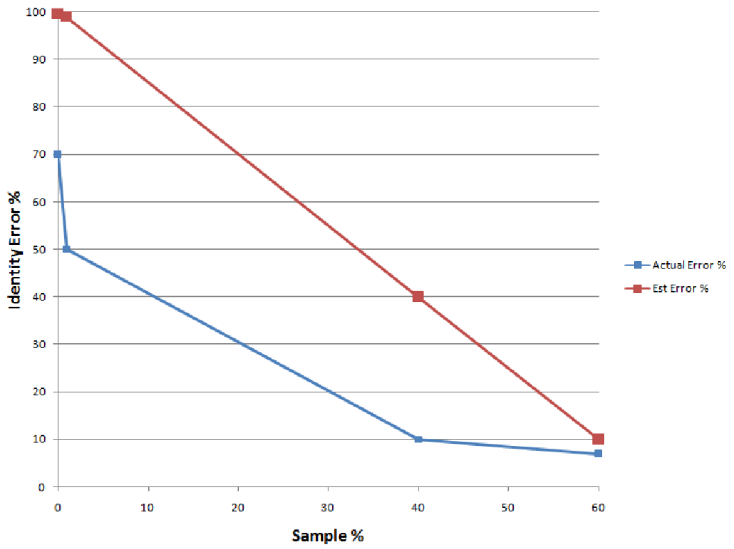
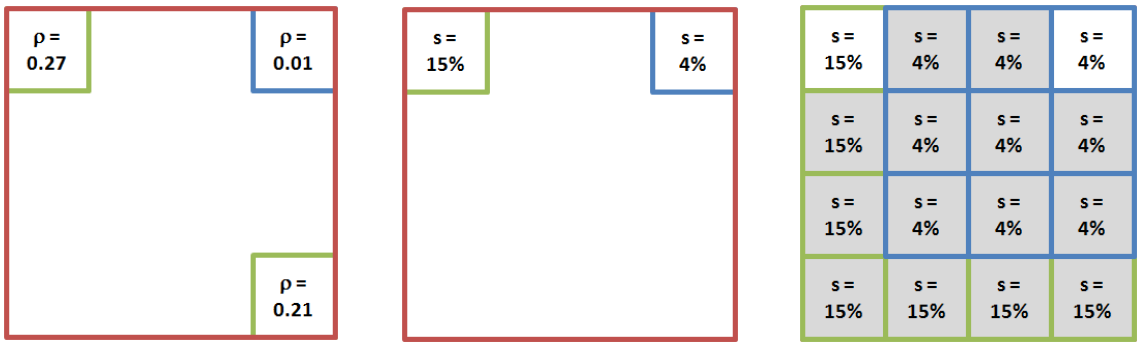


Figure 3.5: S-EST: RS_NN Overhead Estimator

plan-location error typically lags behind the plan-identity error. However, it is possible that the user may have given a significantly larger tolerance for identity error as compared to location error, in which case the assumption is no longer valid. To handle this situation, we artificially treat the lower of the tolerance levels to be the plan-identity error in computing the time estimation, in the process achieving a conservative estimate (note that this assumption is only intended for generation time estimation and does not impact the approximation process itself). In the current implementation, the seed set consists of 50 query points, which typically takes less than a minute to optimize on standard platforms.

G-EST: The GS_PQO estimator is known as G-EST. It concentrates on all the corner rectangles except that near the origin of the query space. First, the corner points of those rectangles are optimized and plan-richness factor is calculated for all of them. Among the rectangles present near axis, we choose one with highest plan richness factor for further grid partitioning considering user given error threshold as the bound. We evaluate ρ_t as mentioned in Section 3.2.2.1, using this information for the corner rectangle and keep on optimizing until the bound is reached. The rectangle at the highest selectivity region is also optimized according to the user given error thresholds using the grid partitioning



(a) **Initial Rectangles** (b) **Rectangle chosen for optimizations** (c) **Sample % after optimizations**

Figure 3.6: Execution of GS_PQO Overhead Estimator

mechanism. Now we consider that each rectangle along the axis will consume same number of optimizations as the chosen representative from the axis and the rest of the rectangles will take as much sample as the highest selectivity box. We sum up the sample size accordingly as the final estimate. Figure 3.6 shows the different stages of working of the estimator for a 2D selectivity space. First the corner points of the 3 rectangle are optimized to find out the ρ value, as shown in Figure 3.6(a). Among the rectangles at left-top and right-bottom, the one with higher ρ value is chosen to be processed further along with the right-top box. Figure 3.6(b) shows the % samples consumed by them. The *gray* rectangles of Figure 3.6(c) shows the final assignment to all the rectangles of the selectivity space. The final estimate of required sample size is 8.8% in this case. Note that we avoid considering any rectangle around origin because of two reasons, 1) it consumes a large number of samples and 2) it gives a huge overestimate when same % of samples are assigned to all the rectangles along the axis.

According to our study, we have found the initial box with rectangle edge width $\sqrt{r}$ produces satisfactory estimation.

3.3 Class II Optimizers

In the algorithms described above for the Class I optimizers, we run into situations wherein we are forced to pick from a set of equivalent candidate plans in order to make an assignment for an un-optimized query point. For example, in the RS_NN approach, if there are multiple nearest neighbors at the same distance. Similarly, in the GS_PQO approach, when the ρ value of a rectangle goes below the threshold and there remain unassigned internal points. The strategy followed is to make a random choice from the closest neighboring plans.

Class I	**Class II**
In GS_PQO stopping threshold is set as $\rho_t = \frac{\rho_{max}}{\xi_{max}} \times \theta_I$	Stopping threshold can be relaxed to $(\rho_t \times \sqrt{d})$
Random choice in tie breaking	Perform FPC and choose plan with least cost

Table 3.2: Differences in Class I and Class II Algorithms

For Class II optimizers, however, which offer a "foreign plan costing" (FPC) feature, we can make a more informed selection: Specifically, cost all the candidate plans at the query point in question, and assign it the lowest cost plan. This method significantly helps in reducing the plan-location error, since it enables precise demarcation of the boundaries between plan optimality regions. A direct fallout obtained through our empirical investigation (see Table 3.4) is that the value of ρ_t, can be increased further to $\sqrt{d}$ times than the value used for Class I while still maintaining the same accuracy characteristics, in the process noticeably lowering overheads. The differences in Class I and Class II optimizers working is given in Table 3.2.

Another point to be noted here is that plan-costing is much cheaper than the optimizer's standard optimal-plan-searching process [48], and hence the overheads incurred through costing are negligible compared to those incurred through optimization. In our experience, the overhead ratio of plan-costing to plan-searching is around **1:10** in the commercial optimizers, while in our implementation of this feature in PostgreSQL, it is close to **1:100**.

The improvement proposed above is beneficial as long as the number of points where

FPC is performed does not add significant overhead and at the same time this exercise should have a considerable impact on reducing the plan-location error. In the results shown in Table 3.10 and Table 3.11, we see that both these criteria are met.

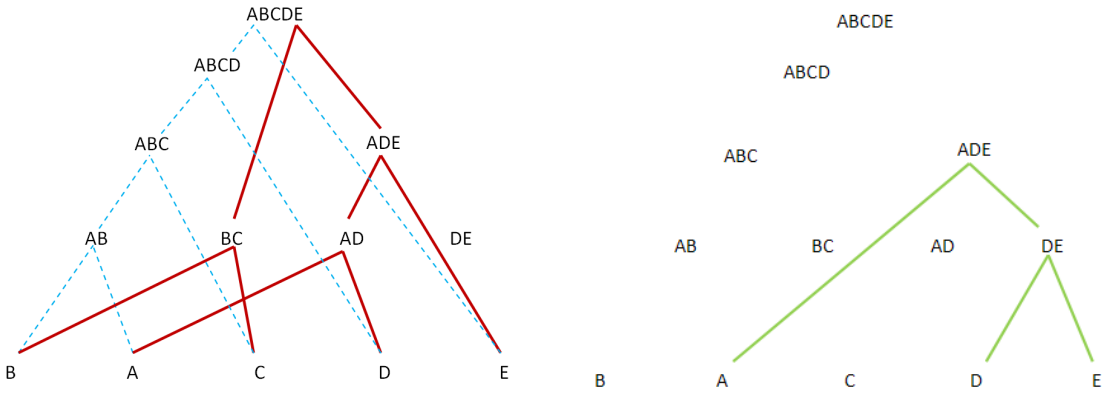
3.4 Class III Optimizers

The algorithms discussed thus far minimize the number of explicit optimizations performed by assuming certain properties of the plan diagram and using these properties to infer plan assignments from the optimized query points. The algorithms proposed for this class of optimizers utilize the foreign-plan-costing (FPC) and plan-rank-list (PRL) features offered by the Class III optimizer API. Specifically, it is assumed that for each query point, the optimizer provides both the best plan and the second-best plan. We now present the algorithm developed for generating PRL.

3.4.1 The Plan Rank List

The query optimizers available in commercial and public-domain database management systems provide only the best query execution plan i.e. the plan which incurs least cost to execute the query. To get the cost-enumerated list of top k plans, the dynamic programming based query optimization needs to be modified. We will discuss these modifications on the query optimization technique defined and illustrated in [68].

Given a query associated with multiple relations, the query optimizer searches for the best query execution plan by finding the best join order for successively larger subsets of relations [68]. Further *Join-Graph* is evaluated to reduce the search space to only meaningful join orders. At each step, sub-plans having neither least cost nor some interesting order [68] are pruned. After termination of search process, the least cost plan from this search tree is chosen. Generating the Plan Rank List at first glance from this search tree seems to be obvious and trivial - sort all the plans available at the root according to their cost. Unfortunately, this will not work as some plans may get *pruned*



(a) **Path structure of top 2 plans retrieved at root in a DP Lattice**

(b) **Pruned sub-plan of a candidate to 2nd best plan**

Figure 3.7: Structure of DP Lattice

early in the search process while being compared to the best plan and may not appear in the final search tree. Figure 3.4.1 shows a sample DP-Lattice. The path structure of best plan (solid-red) and its 2nd sibling (dotted-blue) which reached the root i.e. $ABCDE$ of the DP lattice are shown in the Figure 3.7(a). A candidate best plan e.g. the sub-plan shown in Figure 3.7(b) got pruned earlier. Here it looks like that the node ADE pruned the join order $A \bowtie DE$ as compared to $AD \bowtie E$ (best plan) for being costlier.

In this thesis we concentrate only on getting PRL with $k = 2$.

Additive Second Best Plan Search. We propose an addition to the current data structure of a node, which enables a node to store cheapest two sub-plans generating it and propagate both to the higher nodes in the lattice. As shown in Figure 3.8, the dotted lines depict the path of the 2nd cheapest plans at the respective node e.g. now both the paths for generating the node ADE i.e. $AD \bowtie E$ and $A \bowtie DE$ are saved. In this process the 2nd best plan will be explicitly found at the root node. One of the major issue with this approach is that it would now require *four* times the optimization time at each node. For example, at node $ABCDE$, we would individually compute the best combination strategy for $(AD \bowtie E) \bowtie BC$, for $(AD \bowtie E) \bowtie CB$, for $(A \bowtie DE) \bowtie BC$,

and for $(A \bowtie DE) \bowtie CB$, considering CB is the 2nd best way of generating $(B \bowtie C)$.

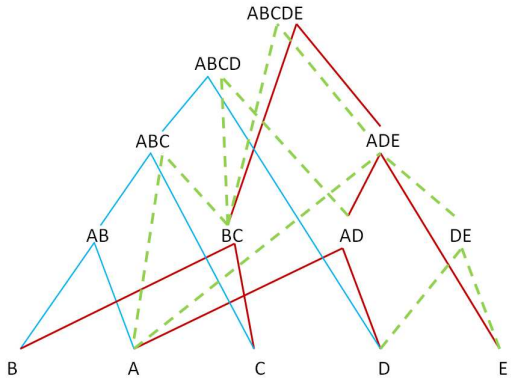


Figure 3.8: Additive Second Best Plan Search Algorithm

However, we can optimize on the above by realizing that the best combination strategy will be the *same* for all four options i.e. the best join strategy and join order identified for $(AD \bowtie E) \bowtie BC$ and $(A \bowtie DE) \bowtie BC$ will be same. This is because the strategies are evaluated in terms of cardinalities and number of distinct values present in BC and ADE, which is independent of the underlying sub-plan generating them. Therefore, only one optimization, say $(AD \bowtie E) \bowtie BC$, needs to be carried out, and the results reused for the other three pairings. All though this method of inheriting cost of best-pair can not be applied directly to sub-plans with interesting orders. But practically for TPC-H templates we have seen very few plans with interesting order, which does not affect the run-time significantly.

We have implemented this algorithm in OptPub and implementation details are given in Chapter 6. We have observed that increase in space for modifying the data structure used for representing one node and retaining the sub-plans went up to maximum $10MB$ (the original data structure takes roughly $6MB$), which is quite reasonable with current systems. This results are obtained with TPC-H benchmark queries, where the maximum node in a lattice was less than 50. Note that we keep twice as much path (1 KB) but same number of nodes (124 KB). The time for retrieving PRL with $k = 2$ was seen to be taking roughly *10%* more time compared to simple optimization.

But, even this approach is wasteful in terms of memory since we need to carry around the extra baggage of the second-best plan throughout the DP lattice.

Iterative Second Best Plan Search. A much more efficient approach is the following: Compute the best plan using the standard DP mechanism with the only minor modification being that at the top node in the lattice, in addition to the best plan (P^I_{top}), temporarily store the second-best plan (P^{II}_{top}) and its cost (c^{II}_{top}). Now, observe that the genuine second-best plan is either P^{II}_{top} itself, or if there has to be an intermediate plan between P^I_{top} and P^{II}_{top}, then that plan must have been pruned *along the path of* P^I_{top}. Therefore, it is sufficient to now traverse *only that path*, starting with the leaves and propagating the second-best sub-plan to higher nodes in the lattice. At each node along this path, if the cost of the second-best sub-plan happens to exceed c^{II}_{top}, then we can immediately terminate the process, and proclaim P^{II}_{top} to be the second-best plan. However, if the second-best sub-plan reaches the root node and its cost is less than c^{II}_{top}, then it is declared the second-best plan.

Note that when we say the path of best-plan, we actually mean the node-path and all alternative inputs along the node-path need to also be explored. That is, just like the second-best plan at top node need not be second-best plan globally, similarly, the second-best plan at each node along the best-plan node path has to be based on considering all inputs to these nodes, and not just the input path followed by the best-plan.

We now move on to presenting for the Class III optimizers. Both the algorithms work following similar approach to the popular Flood-Fill [75] algorithm used in graphics to paint a particular bounded area with a color. In our case we paint the region with a plan. The *PlanFill* algorithm can be used to efficiently generate *completely accurate* plan diagrams. Subsequently, we provide a variant, *Relaxed-PlanFill* algorithm, which trades error, based on the user's bound, for reduction in optimization effort.

3.4.2 The *PlanFill* Algorithm

The *PlanFill* algorithm for a 2D query template is shown in Algorithm 3. The algorithm starts with optimizing the query point $q(x_{min}, y_{min})$ corresponding to the bottom-left query point in the plan diagram. Let p_1 be the optimizer-estimated optimal plan at q, with cost $c_1(q)$, and let p_2 be the *second best* plan, with cost $c_2(q)$. We then assign the plan p_1 to all points q' in the *first quadrant* relative to q as the origin, which obey the constraint that $c_1(q') \leq c_2(q)$. After this step is complete, we then move to the next unassigned point in row-major order relative to q, and repeat the process, which continues until no unassigned points remain.

This algorithm is predicated on the *Plan Cost Monotonicity* (PCM) assumption that the cost of a plan is monotonically non-decreasing throughout the selectivity space, which is true in practice for most query templates [18].

The following theorem proves that the *PlanFill* algorithm will exactly produce the true plan diagram P without any approximation whatsoever. That is, *by definition*, there are no plan-identity and plan-location errors.

Theorem 1 *The plan assigned by* PlanFill *to any point in the approximate plan diagram* A *is exactly the same as that assigned in* P.

Proof: Let $P_o \subseteq \mathsf{P}$ be the set of points which were optimized. Consider a point $q' \in \mathsf{P} \setminus P_o$ with a plan p_1. Let $q \in P_o$ be the point that was optimized when q' was assigned the plan p_1. Let p_2 be the second best plan at q.

For the sake of contradiction, let p_k ($k \neq 1$), be the optimal plan at q'. We know that for a cost-based optimizer, $c_k(q') < c_1(q')$. This implies that $c_k(q') < c_2(q)$ (due to the algorithm). Using the PCM property, we have $c_k(q) \leq c_k(q') \Rightarrow c_1(q) \leq c_k(q) < c_2(q)$. This means that p_2 is not the second best plan at q, a contradiction. ■

3.4.3 The *Relaxed-PlanFill* Algorithm

While *PlanFill* always ensures zero error, we now investigate the possibility of whether it is possible to utilize the permissible error bound of θ to further reduce the computational

Algorithm 3 The *PlanFill* Algorithm

PlanFill (QueryTemplate QT)

```
1: Let A be an empty plan diagram.
2: q ← (x_min, y_min)
3: while q ≠ null do
4:    Optimize query template QT at point q.
5:    Let p_1 and p_2 be the optimal and second-best plan at q, respectively.
6:    for all unassigned points q' in the first quadrant of q do
7:       if c_1(q') ≤ c_2(q) then
8:          assign plan p_1 to q'
9:       end if
10:   end for
11:   q ← next unassigned query point in A
12: end while
13: return A
```

overheads of *PlanFill*. To this end, we propose the following *Relaxed-PlanFill* algorithm: The plan assignment constraint $c_1(q') \leq c_2(q)$ is relaxed to be $c_1(q') \leq (1+\gamma)c_2(q)$ with $(\gamma > 0)$, resulting in fewer optimizations being required to fully assign plans in the diagram.

The choice of γ is a function of θ and μ, the slope of the cost function c_2 at q with regard to maximum cost value of the plan diagram. The reason behind choosing μ is to make γ tighter with increasing selectivity as around lower selectivity region the ratio between the cost values of two neighboring plans is found to be higher than their higher selectivity counterpart at the same distance. We use slope of the cost function to capture this gap between costs at different selectivity region. Our empirical assessment indicates that setting $\gamma = \mu * \theta$ (e.g. with $\theta = 10\%$,and $\mu = 0.1$, $\gamma = 0.01$) is sufficient to meet the error requirements and simultaneously significantly reduce the overheads. For example, $\theta = 10\%$ can be achieved with only around **1%** overheads, as seen in the following section. For cases where $\theta_I \neq \theta_L$, we assume $\theta = \min\{\theta_I, \theta_L\}$.

3.5 Experimental Results

The testbed used in our experiments is the Picasso optimizer visualization tool [76], executing on a Sun Ultra 20 workstation equipped with an Opteron Dual Core 2.5GHz processor, 4 GB of main memory and 720 GB of hard disk, running the Windows XP Pro operating system. The experiments were conducted over plan diagrams produced from a variety of two, three, and four-dimensional **TPC-H** [83] and **TPC-DS** [82] based query templates. In our discussion, we use QTx to refer to a query template based on Query x of the TPC-H benchmark, and DSQTx to refer to a query template based on Query x of the TPC-DS benchmark. The TPC-H database was of size 1GB, while the TPC-DS database occupies 100GB. The plan diagrams were generated with a variety of industrial-strength database query optimizers – we present representative results here for a commercial optimizer anonymously referred to as OptCom, and a public-domain optimizer, referred to as OptPub. We will present results with query points being uniformly distributed over the selectivity space unless explicitly mentioned.

In the remainder of this section, we evaluate the various approximation strategies with regard to their computational efficiency, given user-specified bounds for plan-identity and plan-location error. For simplicity of exposition, we will assume in the sequel that users specify the same bound $\theta_I = \theta_L = \theta$ on both metrics, however our methods are applicable even if $\theta_I \neq \theta_L$. The bounds we consider here are $\theta = 10\%$ and $\theta = 1\%$.

Our analysis was carried out over an extensive suite of query templates. However, we selectively present results here for "challenging" plan diagrams that feature a sufficiently rich set of plans (≥ 10 plans). This is due to fact that with < 10 plans the notion of 10% error becomes equivalent to 0% error.

3.5.1 Class I Optimizers

We will first show the behavior of the three estimators discussed in Section 3.2.1 to validate our choice of estimator. Then we present the effect of different stopping conditions imposed on GS_PQO and how it influences the number of plans discovered inside a rectangle.

Dimension/ Resolution	Query Templates	$\hat{\alpha}_{max}$		$\hat{\alpha}_{GEE}$		$\hat{\alpha}_{hybrid}$	
		s (%)	ϵ_I (%)	s (%)	ϵ_I (%)	s (%)	ϵ_I (%)
2D: 300 x 300	QT2	53	5	23	18	36	8
	QT3	30	5	1	27	25	5
	QT4	5	0	1	20	5	0
	QT5	8	10	6	13	8	10
	QT7	19	0	12	0	17	0
	QT8	56	7	35	10	35	10
	QT9	55	6	33	10	33	10
	QT20	37	4	26	4	28	4
	QT21	29	2	18	4	18	4
2D: 1000 x 1000	QT8	47	5	25	8	25	8
	QT21	15	8	5	12	11	10
3D: 100 x 100	QT8	45	6	20	12	25	11
	QT9	47	7	30	11	35	8

Table 3.3: Comparative study on estimator performance $\hat{\epsilon}_I = 10\%$

Performance of different ϵ_I estimators. Table 3.3 shows the performance of the estimators $\hat{\alpha}_{max}$, $\hat{\alpha}_{GEE}$ proposed in [7] along with the hybrid estimator ($\hat{\alpha}_{hybrid}$) combining the former two estimators, as discussed in Section 3.2.1. $\hat{\alpha}_{GEE}$ when used as stopping criterion terminates quite early without reaching the goal of $\theta_I = 10\%$ for the 2D query templates QT2,QT3 and QT4 with resolution 300. Again, $\hat{\alpha}_{max}$ turns out to be too conservative in many cases e.g. QT21,QT9 and QT20. However, the performance of hybrid estimator $\hat{\alpha}_{hybrid}$ was found to be satisfactory almost always. All the experiments performed for RS_NN later in this chapter relies on the $\hat{\alpha}_{hybrid}$ estimator.

Behavior of ρ_t in GS_PQO. This set of experiments are to validate the "micro-PQO" assumption. We study the number of plans missed inside a rectangle R, by running GS_PQO for different ρ_t values. The initial rectangle size for which we calculated ρ was set to $\sqrt{(r)}$. The experimental results are shown in Table 3.3. For none of the cases we found "micro-PQO" to be violated. Furthermore, the results suggests that ϵ_I contribution of rectangles with $\rho = 0.1$ and $\rho = 0.2$ are almost similar. This observation later helped us to relax the stopping condition of GS_PQO for Class II optimizers.

We now move on to presenting comparative analysis of algorithms developed for Class I optimizers. We start with evaluating the performance of the two algorithms applicable to Class I optimizers, namely, RS_NN and GS_PQO. In the RS_NN algorithm, as mentioned

Dimension/ Resolution	Query Template	Plans #	Max plans missed in a rectangle with $\rho \leq \rho_t$				
			$\rho_t = 0$	$\rho_t = 0.05$	$\rho_t = 0.1$	$\rho_t = 0.15$	$\rho_t = 0.2$
2D: 100X100	QT2	44	0	0	0	0	0
	QT3	16	0	0	0	0	0
	QT4	11	0	0	0	0	0
	QT5	23	0	0	0	2/7	2/7
	QT7	12	0	0	0	0	1/5
	QT8	50	0	0	0	0	0
	QT9	44	0	0	1/7	1/7	1/7
	QT10	17	0	0	0	1/5	1/5
	QT11	16	0	0	2/6	2/6	2/6
	QT16	32	0	0	0	1/5	3/8
	QT17	12	0	0	0	0	0
	QT18	8	0	0	0	0	0
	QT20	33	0	0	0	4/9	4/16
	QT21	42	0	0	1/8	3/8	3/8
2D: 300X300	QT2	76	0	0	1/7	1/7	1/7
	QT3	22	0	0	0	4/8	4/8
	QT4	12	0	0	0	0	0
	QT5	31	0	1/4	1/4	1/4	4/7
	QT7	17	0	1/4	1/5	2/7	7/9
	QT8	42	0	0	0	1/9	1/9
	QT9	41	0	0	0	2/9	2/9
	QT10	31	0	0	3/6	3/9	3/9
	QT11	20	0	1/4	4/9	4/9	4/9
	QT16	38	0	0/7	0	4/9	4/9
	QT17	12	0	0	0	0/7	0/7
	QT18	8	0	0	0	0/3	0
	QT20	46	0	2/7	4/7	5/12	5/12
	QT21	48	0	1/3	1/7	3/9	3/9
2D: 1000X1000	QT8	132	0	2/4	4/13	4/13	7/13
	QT16	25	0	0	0	0	0
	QT21	58	0	0	6/10	6/10	8/14
3D: 100X100X100	QT8	190	0	0	0	0	1/13
	QT9	404	0	0	2/16	2/23	6/33
	QT21	130	0	0	0	1/14	3/14

Table 3.4: Quality of ρ_t as an estimator [OptCom- TPC-H database]

earlier, the parameter δ, which specifies the transition of the algorithm from Stage 1 to Stage 2, is set to 0.3, while the sample size increments are 1% of the space. For the GS_PQO algorithm, the resolution of the initial grid along each dimension is set to a value around $\sqrt{resolution}$, at which the plan diagram is to be generated. As an example, to approximate a 2D plan diagram with 300×300 resolution, we set the initial sample size of RS_NN to 900 and the initial grid of GS_PQO to 16×16.

Error Bound = 10%. For the above framework, Table 3.5 shows the algorithmic efficiency of the RS_NN and GS_PQO algorithms relative to the brute-force exhaustive approach for a variety of multi-dimensional query templates, under a $\theta = 10\%$ constraint. The efficiency is presented both in terms of actual time, as well as in terms of the number of optimizations that were carried out. The bracketed numbers in the *TimeTaken* columns

Dim/ Res	Query Temp--late	Pl-ans #	Gen Time	Approximation Time Taken		Samples s(%)		Error (%)			
								RS_NN		GS_PQO	
				RS_NN	GS_PQO	RS_NN	GS_PQO	ϵ_I	ϵ_L	ϵ_I	ϵ_L
2D: 100X100	QT2	44	30 m	10 m(33%)	3 m(10%)	33%	9%	10%	2%	0%	9%
	QT3	16	8 m	3.4 m(42%)	45 s(9%)	42%	9%	10%	6%	6%	10%
	QT4	11	6 m	1 m(16%)	47 s(13%)	16%	13%	0%	10%	0%	6%
	QT5	23	45 m	7 m(15%)	5 m(12%)	15%	12%	4%	9%	0%	6%
	QT7	12	38 m	1 m(42%)	4 m(10%)	42%	10%	6%	5%	0%	6%
	QT8	50	1 h	22 m(38%)	9 m(15%)	38%	15%	10%	7%	6%	10%
	QT9	44	2 h	45 m(39%)	15 m(13%)	39%	13%	10%	6%	2%	9%
	QT10	17	12 m	1 m(9%)	1 m(8%)	9%	8%	6%	7%	0%	5%
	QT11	26	36 m	4 m(10%)	3 m(8%)	10%	8%	6%	5%	0%	8%
	QT16	32	10 m	2 m(21%)	1 m(13%)	21%	12%	9%	10%	3%	13%
	QT17	12	21 m	1 m(6%)	1 m(6%)	6%	6%	9%	8%	0%	7%
	QT18	8	1 h	12 m(20%)	3 m(5%)	20%	5%	9%	7%	0%	6%
	QT20	33	4 h	1.6 h(40%)	16 m(7%)	40%	7%	3%	5%	3%	8%
	QT21	84	30 m	4 m(13%)	5 m(14%)	13%	14%	10%	10%	0%	7%
2D: 300X300	QT2	76	4.3 h	1 h(25%)	13 m(5%)	25%	5%	8%	4%	5%	8%
	QT3	22	1.7 h	30 m(25%)	7 m(7%)	25%	7%	5%	8%	0%	5%
	QT4	12	1 h	3 m(5%)	4 m(6%)	5%	6%	0%	10%	0%	6%
	QT5	31	8.3 h	40 m(8%)	23 m(5%)	8%	5%	10%	5%	0%	6%
	QT7	17	6 h	1 h(17%)	24 m(4%)	17%	4%	0%	3%	6%	3%
	QT8	92	11 h	3.6 h(35%)	43 m(7%)	35%	7%	10%	3%	1%	7%
	QT9	91	1.1 d	9 h(34%)	1.5 h(6%)	34%	6%	10%	3%	2%	6%
	QT10	31	5 h	30 m(10%)	9 m(3%)	10%	3%	10%	3%	6%	4%
	QT11	20	2.5 h	8 m(5%)	4 m(3%)	5%	3%	5%	6%	0%	9%
	QT16	38	1.6 h	8 m(8%)	5 m(5%)	8%	5%	5%	10%	0%	11%
	QT17	12	2.5 h	8 m(5%)	6 m(2%)	5%	2%	0%	3%	0%	4%
	QT18	8	7 h	21 m(5%)	6 m(2%)	5%	2%	10%	3%	0%	6%
	QT20	46	1.3 d	8.7 h(28%)	37 m(2%)	28%	2%	4%	2%	11%	8%
	QT21	48	5 h	1 h(18%)	38 m(7%)	18%	7%	4%	4%	7%	2%
2D:1000 X 1000	QT8	132	6 d	29 h (21%)	3.3 h(2%)	21%	2%	3%	10%	11%	5%
	QT9	125	10 d	3 d(30%)	4 h(2%)	30%	2%	4%	5%	2%	15%
	QT21	58	2.2 d	8.6 h(16%)	47 m(1%)	16%	1%	10%	2%	0%	6%
3D:100X 100X100	QT8	190	6.5 d	2 d(27%)	17 h(12%)	27%	11%	1%	2%	1%	9%
	QT9	404	10 d	3 d(30%)	1.3 d(13%)	30%	12%	7%	10%	3%	9%
	QT21	130	3 d	1.2 d(37%)	10 h(14%)	37%	13%	1%	4%	1%	7%
3D:300X 300X300	QT8	314	4 mons	–	2.6 d(2%)	–	2%	–	–	–	–
4D:30X 30X30X30	QT8	243	5 d	23 h(19%)	15 h(12%)	19%	12%	12%	9%	4%	9%

Table 3.5: Class I : Efficiency with TPC-H ($\theta = 10\%$) [OptCom]

indicate the percentage time taken relative to the exhaustive approach.

We see in Table 3.5 that the RS_NN algorithm requires a substantial amount of time, or equivalently, number of optimizations, to generate the approximate plan diagram. For example, with the 3D QT9 template at a resolution of 100 per dimension, RS_NN takes about 30% of the exhaustive time. On the other hand, GS_PQO exhibits a much better performance, requiring only 13% overheads – in fact, our experience has been that it needs less than 15% of the exhaustive time *across all templates*. Moreover, as can be seen from Table 3.5, we have also produced an approximate plan diagram for the 3D QT8 template at a resolution of 300 per dimension, corresponding to *27 million query points* in only 2.6

Dim/ Res	Query Temp- -late	Pl- ans #	Gen Time	Approximation Time Taken		Samples s(%)		Error (%)			
								RS_NN		GS_PQO	
				RS_NN	GS_PQO	RS_NN	GS_PQO	ϵ_I	ϵ_L	ϵ_I	ϵ_L
2D: 100X100	DSQT12	13	16 m	4 m(25%)	28 s(3%)	25%	3%	12%	3%	0%	0%
	DSQT17	39	8 h	2.6 h(39%)	39 m(8%)	39%	8%	8%	8%	10%	9%
	DSQT18	47	3.5 h	1.4 h(40%)	20 m(10%)	40%	10%	6%	7%	2%	9%
	DSQT19	36	2 h	24 m(20%)	11 m(9%)	20%	9%	3%	3%	8%	9%
	DSQT25	33	8 h	4.6 h(65%)	41 m(9%)	65%	9%	10%	9%	9%	10%
	DSQT25a	51	8 h	1.5 h(24%)	1 h(12%)	24%	12%	12%	11%	0%	11%
	DSQT25b	45	7.3 h	2.6 h(36%)	30 m(7%)	36%	7%	9%	9%	0%	8%
	DSQT50	10	1 h	20 m(18%)	4 m(7%)	30%	7%	8%	9%	0%	1%
	DSQT76	18	1.5 h	31 m(34%)	14 m(15%)	34%	15%	8%	2%	11%	10%
2D: 300X300	DSQT12	15	2.2 h	40 m(29%)	2 m(2%)	29%	2%	7%	4%	11%	5%
	DSQT18	81	1 d	9 h(38%)	1 h(4%)	38%	4%	10%	11%	5%	7%
	DSQT19	42	17 h	1 h(7%)	34 m(4%)	7%	4%	7%	7%	2%	6%
	DSQT29	37	3 d	11 h(15%)	3.2 h(4%)	15%	4%	2%	3%	3%	7%
3D:100X 100X100	DSQT19	167	10 d	2 d(20%)	1 d(14%)	20%	14%	2%	8%	1%	8%

Table 3.6: Class I : Efficiency with TPC-DS ($\theta = 10\%$) [OptCom]

days with GS_PQO – the estimated generation time with the brute-force approach is 4 months! The difference in percentage value between optimization and approximation time taken is contributed by three factors e.g. 1) time to complete the NN inference step and 2) the iterative ϵ_L estimation phase excluding the optimization time. For GS_PQO the time required to build and maintain different maxHeap data structure is also included. An interesting point to note is that for RS_NN the optimization percentages are virtually identical to the time percentages for most of the query templates. This is because ϵ_L lags ϵ_I for all of them except some complex diagram like QT21. Even for GS_PQO these extra times do not seem to add any significant overheads.

We see that the estimators designed for RS_NN and GS_PQO almost always result in meeting the user's error bounds or being in their close proximity.

Turning our attention to Table 3.6, which repeats the above experiment on the TPC-DS database, we see that the results are even more striking. RS_NN incurs large overheads in general, typically around 40%, whereas GS_PQO again does not exceed 15%.

Error Bound = 1%. When the user's error constraint is tightened from 10 percent to 1 percent, the resulting algorithmic performance is shown in Table 3.7. Only GS_PQO is shown since for this stringent constraint, the RS_NN algorithm tends to optimize almost the entire space. Further to make the 1% error bound meaningful, we have considered only plan diagrams having around or over 100 plans. It can be seen from the table that

Dim/ Res	Query Template	Plans #	Exhaustive Gen time	Approxima- -tion Time	Samples (%)	Error (%)	
						ϵ_I	ϵ_L
2D: 300X300	QT 8	92	10.5 h	3.6 h (35%)	35 %	0 %	0.25 %
	QT 9	91	1 d 3 h	7.3 h (30%)	30 %	0 %	2 %
2D:1000 X1000	QT 8	132	6 d	1.8 d (30%)	30 %	2 %	1 %
3D:100X 100X100	QT 8	190	6.5 days	1.6 d (25%)	25 %	0.5 %	1.5 %
	QT 9	404	10 d	4 d (40%)	40 %	1 %	1 %
	QT 21	130	3 d	11 h (16%)	16 %	0.77 %	1.5 %

Table 3.7: Class I : Efficiency with TPC-H ($\theta = 1\%$) [OptCom]

Dim/ Res	Query Template	No. of Plans	Samples (%)		RS_NN		GS_PQO	
			RS_NN	GS_PQO	ϵ_I	ϵ_L	ϵ_I	ϵ_L
2D: 100X100	QT2	44	32 %	6.4 %	7 %	8 %	4.4 %	9 %
	QT3	16	18 %	6.4 %	4 %	10 %	4.4 %	9 %
	QT4	11	14 %	12.6 %	10 %	7 %	0 %	7.9 %
	QT5	23	32 %	17.6 %	9 %	10 %	10.8 %	10.9 %
	QT7	12	22 %	12 %	9 %	5 %	3 %	3 %
	QT8	50	42 %	10.6 %	10 %	11 %	3.7 %	6.8 %
	QT9	44	33 %	9.3 %	4 %	7 %	0 %	8.2 %
	QT10	17	19 %	3.8 %	4 %	10 %	0 %	7.4 %
	QT11	16	14 %	18 %	8 %	9 %	0 %	1.1 %
	QT16	32	16 %	8.5 %	10 %	10 %	0 %	5.9 %
	QT17	12	25 %	14%	7 %	9 %	2 %	9%
	QT18	8	16 %	4.5 %	8 %	7 %	9.1 %	8.4 %
	QT20	33	31 %	19 %	7 %	10 %	0 %	7.8 %
	QT21	42	31 %	9.5 %	6 %	8 %	0 %	8.2 %
2D: 300X300	QT5	31	61 %	4.9 %	0 %	3 %	7.7 %	9.9 %
	QT8	92	27 %	13 %	10 %	11 %	5.7 %	10.6 %
	QT9	91	39 %	11.3 %	3 %	5 %	5.2 %	8.5 %
	QT21	48	5 %	4.7 %	0 %	1 %	0 %	3.9 %

Table 3.8: Class I : Efficiency for Exponential Distribution ($\theta = 10\%$) [OptCom]

by optimizing around 40% of the points, GS_PQO is able to generate extremely accurate approximate plan diagrams.

Portability on Exponential Diagrams The earlier experiments were performed on plan diagrams which have the query points uniformly distributed over the relational selectivity space. Now we show portability of our algorithms for plan diagrams with *exponential* query point distribution. With the *exponential* distribution, the density of points is maximum near the origin and becomes progressively lower moving outwards in the space. The motivation for the *exponential* distribution stems from the observation in the literature [47, 48, 62, 63, 64] that plan density is often high around the origin and along the axes, and it may therefore be useful, from a computational perspective, to focus the query workload on these regions. Table 3.8 presents performance of both RS_NN and GS_PQO algorithm on some of the complex TPC-H query templates with exponential distribution.

The results suggests these algorithms are also applicable for exponential diagrams, with around 5% increase in optimization overheads than its uniform counterpart. The increase in overhead was expected as in case of exponential diagrams taming identity error itself requires high sample size. The value of ϵ_L was measured in similar fashion as is done in case of uniform query point distribution.

Estimator Performance Our next experiment studies the quality of the *overhead estimates* provided by the estimators. The results shown in Table 3.9 indicates the estimation quality of S-EST and G-EST with regard to actual approximation time taken by RS_NN and GS_PQO respectively. We see here that, in general S-EST produces *conservative* estimates, whereas estimations produced by G-EST are almost always close to the actual approximation overheads.

3.5.2 Class II Optimizers

We now move on to demonstrate how the FPC feature, provided by Class II optimizers, can be used to improve the performance of GS_PQO. Tables 3.10 and 3.11 show the effort required by GS_PQO for obtaining approximate plan diagrams with $\theta = 10\%$ on the TPC-H and TPC-DS benchmarks, respectively. The "FPC (%)" column in both Tables 3.5 and 3.6 indicates the percentage of FPC performed during the interpolation phase to resolve ties. As can be seen from the results, the percentage FPC performed is low compared to actual optimizations. Further we see here that GS_PQO often reduces the approximation overheads by a significant fraction as compared to the corresponding numbers in Tables 3.5 and 3.6, testifying to the utility of FPC. As a case in point, the 13% overhead incurred by the 3D:100x100x100 flavor of QT9 with the Class I optimizer is reduced to 8.5% with the Class II optimizer.

Another point to note is that the average value of ϵ_I increases for Class II optimizers since we relax the value of ρ_t.

With an error bound of 1%, however, the role of FPC becomes limited since inference is rare, and therefore the diagram approximation time is similar to that seen for Class I

Dim/ Res	Query Template	Plans #	Actual Gen Time	RS_NN Time Taken	S-EST Estimation	GS_PQO Time Taken	G-EST Estimation
2D: 100X100	QT2	44	30 m	10 m	15 m	3 m	5 m
	QT3	16	8 m	3.4 m	7 m	45 s	2 m
	QT4	11	6 m	1 m	3 m	47 s	¡ 1 m
	QT5	23	45 m	7 m	20 m	5 m	6 m
	QT7	12	38 m	1 m	16 m	4 m	9 m
	QT8	50	1 h	22 m	32 m	9 m	8 m
	QT9	44	2 h	45 m	1.2 h	15 m	20 m
	QT10	17	12 m	1 m	30 m	1 m	¡ 1 m
	QT11	26	36 m	4 m	20 m	3 m	4 m
	QT16	32	10 m	2 m	5 m	1 m	¡ 1m
	QT17	12	21 m	1 m	9 m	1 m	5 m
	QT18	8	1 h	12 m	24 m	3 m	11 m
	QT20	33	4 h	1.6 h	2.2 h	16 m	31 m
	QT21	84	30 m	4 m	14 m	5 m	6 m
2D: 300X300	QT2	76	4.3 h	1 h	1.8 h	13 m	26 m
	QT3	22	1.7 h	30 m	1 hr	7 m	15 m
	QT4	12	1 h	3 m	20 m	4 m	3 m
	QT5	31	8.3 h	40 m	3 h	23 m	36 m
	QT7	17	6 h	1 h	2.5 h	24 m	1 h
	QT8	92	11 h	3.6 h	5.5 h	43 m	1.2 m
	QT9	91	1.1 d	9 h	16 h	1.5 h	2.6 h
	QT10	31	5 h	30 m	2 h	9 m	11.3 m
	QT11	20	2.5 h	8 m	1 h	4 m	12 m
	QT16	38	1.6 h	8 m	20 m	5 m	8 m
	QT17	12	2.5 h	8 m	1 h	6 m	30 m
	QT18	8	7 h	21 m	3 h	6 m	25 m
	QT20	46	1.3 d	8.7 h	12 h	37 m	2.8 h
	QT21	48	5 h	1 h	2.2 h	38 m	26 m
2D:1000 X 1000	QT8	132	6 d	29 h	2 d	3.3 h	4.2 h
	QT9	125	10 d	3 d	4 d	4 h	13 h
	QT21	58	2.2 d	8.6 h	1 d	47 m	58 m
3D:100X 100X100	QT8	190	6.5 d	2 d	3 d	17 h	23 h
	QT9	404	10 d	3 d	4.2 d	1.3 d	2.5 d
	QT21	130	3 d	1.2 d	1.8 d	10 h	16 h
4D:30X 30X30X30	QT8	243	5 d	23 h	20 h	15 h	22 h

Table 3.9: Performance of Estimators with TPC-H ($\theta = 10\%$) [OptCom]

optimizers (Table 3.7).

3.5.3 Class III Optimizers

Turning our attention to Class III optimizers, we now evaluate the two algorithms, *PlanFill* and *Relaxed-PlanFill* for TPC-H and TPC-DS benchmark queries. For this experiment, the OptPub engine was modified to (a) implement the FPC feature internally, and (b) to return the second best plan along with the optimal plan when the "explain"command is executed.

Dim/ Res	Query Templates	Plans #	Gen Time	Time taken by GS_PQO	Samp--les(%)	FPC (%)	GS_PQO (%)	
							ϵ_I	ϵ_L
2D: 100X100	QT2	44	0.5 h	2 m(6%)	6%	9%	6.8%	6.7%
	QT3	16	8 m	40 s(8.6%)	8%	3%	6.3%	5.8%
	QT4	11	6 m	47 s(9.7%)	10%	8%	0.0%	5.8%
	QT5	23	45 m	4 m(8.3%)	8%	8%	0.0%	0.9%
	QT7	12	38 m	3 m(8.2%)	8%	6%	0.0%	1.3%
	QT8	50	1 h	5 m(8.2%)	8%	6%	6.0%	4.3%
	QT9	44	2 h	10 m(8.9%)	9%	8%	2.3%	3.4%
	QT10	17	12 m	44 s(6.1%)	6%	8%	0.0%	0.3%
	QT11	16	36 m	1.2 m(3.4%)	3%	4%	6.3%	7.4%
	QT16	32	10 m	37 s(6.2%)	6%	11%	3.1%	6.3%
	QT17	12	21 m	1 s(5.5%)	5%	9%	0.0%	0.9%
	QT18	8	1 h	3 m(4.6%)	5%	6%	0.0%	2.8%
	QT20	33	4 h	7 m(2.8%)	3%	5%	9.1%	7.8%
	QT21	42	30 m	3 m(8.3%)	8%	9%	4.8%	3.2%
2D: 300X300	QT2	76	9.6 h	6 m(2.2%)	2%	6%	5.3%	7.6%
	QT3	22	1.7 h	6 m(5.8%)	6%	3%	0.0%	4.8%
	QT4	12	1 h	3 m(5.7%)	5%	4%	0.0%	6.1%
	QT5	31	8.3 h	15 m(2.9%)	3%	4%	6.5%	1.3%
	QT7	17	6 h	10 m(2.7%)	3%	4%	5.9%	0.2%
	QT8	92	11 h	22 m(3.5%)	4%	3%	2.2%	3.9%
	QT9	91	1.1d	43 m(2.7%)	3%	2%	4.4%	5.8%
	QT10	31	5 h	6 m(1.8%)	2%	4%	11%	2.9%
	QT11	20	2.5 h	1 m(0.5%)	1%	3%	10%	6.7%
	QT16	38	1.6 h	3 m(2.9%)	3%	11%	2.6%	6.9%
	QT17	12	2.5 h	2 m(0.9%)	1%	3%	0.0%	1.3%
	QT18	8	7 h	6 m(1.5%)	2%	3%	0.0%	3.5%
	QT20	46	1.3 d	13 m(0.7%)	1%	4%	9%	6.2%
	QT21	48	5 h	9 m(3%)	3%	10%	2.1%	1.3%
2D:1000 X1000	QT8	132	6 d	1.6 h(1.1%)	1%	2%	8.2%	3.9%
	QT9	404	10 d	1 h(0.5%)	1%	1%	4.8%	11.6%
	QT21	58	2.2d	15 m(0.5%)	1%	1%	6.9%	1.2%
3D:100X 100X100	QT8	190	6.5d	14.5h(9.5%)	9.1%	2%	1.6%	3.7%
	QT9	404	10 d	21 h(8.5%)	8.3%	1%	7.3%	9.4%
	QT21	130	3 d	6 h(8.5%)	8.0%	2%	1.5%	2.0%
3D:300X 300X300	QT8	314 (GS_PQO)	4 mons (est)	2 d(1.7%)	1.6%	3%	-	–
4D:30X 30X30X30	QT8	243	5 d	12 h(10%)	9.7%	4%	4%	6%

Table 3.10: Class II : Efficiency with TPC-H ($\theta = 10\%$) [OptCom]

PlanFill. The performance results for *PlanFill* are shown in Table 3.12 – due to the change in database engine from OptCom to OptPub, the set of query templates with "challenging" plan diagrams differs as compared to our earlier results. We observe that *PlanFill* usually requires at most 10% optimizations to generate a *completely accurate* plan diagram for all query templates, except those based on TPC-H Query 8 and TPC-DS Query 18 and 19, the reason for which is discussed below. The good performance of *PlanFill* can be attributed to the following: Along with the optimizations being performed at select points, all points are costed exactly once. Further, since the FPC feature is internalized in the optimizer, the ratio of plan-costing to plan-searching is approximately 1:100, making the overheads incurred very small.

Dim/ Res	Query Template	Plans #	Gen Time	Time taken by GS_PQO	Samples (%)	GS_PQO (%)	
						ϵ_I	ϵ_L
2D: 100X100	DSQT12	13	16 m	22 s(2.3%)	2%	0%	0%
	DSQT17	39	6.7 h	31 m(6.6%)	7%	12%	5%
	DSQT18	47	3.5 h	17 m(8.2%)	8%	2%	3%
	DSQT19	36	2 h	9 m(7.5%)	7%	8%	7%
	DSQT25	33	7 h	34 m(7.2%)	7%	9%	4%
	DSQT25a	51	6.5 h	48 m(10%)	10%	0%	11%
	DSQT25b	45	7.3 h	29 m(6.7%)	7%	0%	3%
	DSQT50	10	1 hr	4 m(6.6%)	7%	0%	1%
	DSQT76	18	1.5 h	10 m(12%)	12%	12%	10%
2D: 300X300	DSQT12	15	2.2 h	2 m(1.4%)	1%	11%	3%
	DSQT18	81	1 d	46 m(3.2%)	3%	6%	3%
	DSQT19	42	17 h	23 m(2.3%)	2%	2%	6%
	DSQT29	37	3 d	3 h(3.8%)	4%	3%	4%
3D:100X100X100	DSQT19	167	10 d	20 h(8.5%)	8%	3%	12%

Table 3.11: Class II : Efficiency with TPC-DS ($\theta = 10\%$) [OptCom]

Dim/ Res	Query Template	Plans #	Exhaustive Gen time	Time taken by *PlanFill*	Optimizations by *PlanFill* (%)
2D:1000 X1000	QT5	22	5 h 20 m	4 m (1%)	0.17 %
	QT8	20	6 h 10 m	2 h 47 m (45%)	44 %
	QT9	16	6 h 40 m	40 m (10%)	7.4 %
3D:100X 100X100	QT5	23	5 h 48 m	13m (3%)	2.4 %
	QT8	49	5 h 58 m	2 h 2 m (34%)	32 %
	QT9	22	6 h 45 m	5 m (2%)	0.24 %
4D:30X 30X30X30	QT5	37	4 h 50 m	25 m (8%)	5.8 %
	QT8	62	4 h 30 m	1 hr 18 m (29%)	26 %
	QT9	28	6 h 10 m	7 m (2%)	0.7 %
2D:1000X 1000	DSQT7	17	5 h 24 m	30 m (7.8%)	6 %
	DSQT18	27	6 h 12 m	2 h (34%)	32 %
	DSQT19	79	8 h 31 m	2 h 15 m (26%)	23 %
	DSQT26	24	6 h 42 m	35 m (8.6%)	7 %

Table 3.12: Class III : Zero-error Efficiency[OptPub]

Though an investment of 10% optimizations is usually the order of the day, there are occasional scenarios when the *PlanFill* algorithm requires a substantially larger number of optimizations to generate the plan diagram. Such a situation is seen for QT8,DSQT18 and DSQT19 – the reason is that the cost of the second best plan is extremely close to that of the optimal plan over an extended region. Even though the actual plan switch occurs much later, this close-to-optimal cost causes the algorithm to optimize at frequent intervals as the constraint $c_1(q') \leq c_2(q)$ is easily violated leading to the algorithm "panicking too quickly" and choosing to optimize a large number of unnecessary points.

Relaxed-PlanFill. Turning our attention to the *Relaxed-PlanFill* algorithm, whose performance is presented in Table 3.13 for a 10% error bound, we find that it consistently generates approximate plan diagrams while performing less than 5% optimizations. Fur-

Dim/ Res	Query Templates	Plans #	Gen Time	Approximation Time	Samples (%)	Error (%) ϵ_I	Error (%) ϵ_L
2D:1000 X1000	QT5	22	5 h 20 m	3 m (2.4%)	1.8 %	16.6 %	2.6 %
	QT8	20	6 h 10 m	9 m (9%)	7.9 %	0 %	2.8 %
	QT9	16	6 h 40 m	4 m (3%)	2.1 %	0 %	0.8 %
3D:100X 100X100	QT5	23	5 h 48 m	10 m (3%)	1.7 %	9 %	3 %
	QT8	49	5 h 58 m	17 m (5%)	3.4 %	12 %	0.3
	QT9	22	6 h 45 m	4 m (1%)	0.06 %	18 %	10%
4D:30X 30X30X30	QT5	37	4 h 50 m	20 m (7%)	4.5 %	11 %	1 %
	QT8	62	4 h 30 m	10 m (4%)	1.9 %	6 %	5%
	QT9	28	6 h 10 m	5 m (1%)	0.3 %	18 %	13 %
2D:1000X 1000	DSQT7	17	5 h 24 m	6 m (1.8%)	0.6 %	5.8 %	7.9 %
	DSQT18	27	6 h 12 m	18 m (5%)	4.5 %	3.7%	3.8%
	DSQT19	79	8 h 31 m	42 m (8.2%)	7 %	12 %	10.6%
	DSQT26	24	6 h 42 m	12 m (3%)	1.5 %	8.3 %	9.4 %

Table 3.13: Class III : *Relaxed-PlanFill* Efficiency ($\theta = 10\%$) [OptPub]

ther and very importantly, even for the problematic QT8,DSQT18 and DSQT19, due to the relaxation of the effect of the proximity of the second best plan, the plan diagram is now obtained incurring only a small overhead. Finally, the apparently high identity error of 18% for the 4D QT9 query template is an artifact of the low number of plans (28) in the original plan diagram.

In Table 3.13, the maximum number of plans produced by a query template is only 49 which is much below 100 – therefore, the performance of *Relaxed-PlanFill* for $\theta = 1\%$ is equivalent to that of *PlanFill*, which can be viewed as *Relaxed-PlanFill* with $\theta = 0\%$.

A related point to note is that unlike the Optimizer I and II classes where the time and optimization overheads are virtually identical, here the time overheads are a little more than that of optimization. The reason is that although FPC is very cheap, since it has to be invoked for a very large number of points, a small but perceptible time overhead results.

3.5.4 Cost Increase Due to Approximation

A legitimate concern in generating and using approximate plan diagrams is the following: In the erroneous locations, where a different plan has been assigned as compared to the original diagram, is it possible that the substitute plan's (estimated) cost performance is significantly worse than that of the original choice? Our experience is that the cost increase is only a few percent – this is quantified below in Table 3.14, which shows the

maximum cost increase incurred in the erroneous locations, for a representative set of query templates. The experiments were performed for $\theta = 10\%$ and $\theta = 20\%$, by applying FPC at the erroneous locations and comparing with the actual cost of true plan there.

The average error found for individual diagram is small, in fact according to our observation more than 80% of the erroneous points suffer lower than 10% error. Most of the errors occur near the boundary of larger plans or towards higher selectivity region. Note that the number of observations for TPC-H got reduced from 200 to 50 from $\theta = 10\%$ to $\theta = 20\%$. This is due to the fact that we had to ignore plan diagrams with < 20 plans.

Database	θ_I	**# of Templates)**	**Max Cost error (%)**	**Avg Cost error (%)**
TPC-H	10%	200	27%	2%
TPC-H	20%	50	45%	2.5%
TPC-DS	10%	60	31%	2%
TPC-DS	20%	20	50%	2%

Table 3.14: Cost Increase induced by Approximation

Chapter 4

Cost Diagram Approximation

In this chapter we present methods to generate approximate cost diagrams for the three different classes of optimizers. Our approximation procedure does not involve further optimizations and works after the plan diagram approximation has completed. Essentially, the approximation process is required only for Class I optimizers. The FPC feature supported by Class II and III optimizers is capable of generating the *exact* cost diagram associated with the approximate plan diagram by explicitly costing each inferred point. However, for Class II an extra overhead is added due to this. The algorithms developed for Class III perform costing at each point as a part of the process, hence the cost diagram is granted as a bonus.

Let us consider the true cost of a query point $q(x, y)$ in P is $c_P(q)$ and with our approximation technique the cost value is estimated as $c_A(q)$ in A. We denote $\max(c_P) = c_P(q_n)$ as the maximum cost of a query point in P, which is the cost of the top-right query point q_n assuming that the CDP is not violated. First we will define the absolute and relative error metric $\epsilon_{abs}(c_i)$ and $\epsilon_{rel}(c_i)$ to measure the cost approximation error incurred at point q_i between P and A. We then define the combined metric $\epsilon_(c_i)$ used for the practical purposes. This metric is intended to reduce the large % error artificially

induced by very low cost value.

$$\begin{aligned} \epsilon_{abs}(c_i) &= |c_P(q_i) - c_A(q_i)| \\ \epsilon_{rel}(c_i) &= \frac{|c_P(q_i) - c_A(q_i)|}{c_P(q_i)} \\ \epsilon(c_i) &= \min\{\beta_{abs} \times \epsilon_{abs}(c_i), \beta_{rel} \times \epsilon_{rel}(c_i)\} \end{aligned}$$

We use $\beta_{abs} = 1, \beta_{rel} = 100$. Now ϵ_{MAX} and ϵ_{RMS} as defined in Section 1.5 are calculated as:

$$\begin{aligned} \epsilon_{MAX}(\%) &= \max_{i=1}^{n} \epsilon(c_i) \\ \epsilon_{RMS}(\%) &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\epsilon_{abs}(c_i)}{\max(c_P)}\right)^2} \end{aligned}$$

The normalization with respect to $\max(c_P)$ helps in comparing errors of two different cost diagrams in terms of their quality of approximation.

4.1 Class I optimizers

Using empirical assessment of plan cost behavior, a generalized cost function associated with a query execution plan was proposed recently in [19]. For a d-dimensional selectivity space the cost model defined in [19] is given by:

$$\begin{aligned} Cost(x_1, ..., x_d) &= \sum_{i_1}(a_{i_1}x_{i_1} + b_{i_1}x_{i_1}\log x_{i_1}) \\ &\quad + \sum_{i_1<i_2}(a_{i_1i_2}x_{i_1}x_{i_2} + b_{i_1i_2}x_{i_1}x_{i_2}\log x_{i_1}x_{i_2}) \\ &\quad + ... + a_{12..d}(x_1x_2x_3..x_d) \\ &\quad + b_{12..d}(x_1x_2x_3..x_d)\log(x_1x_2x_3..x_d) + a_0 \end{aligned} \tag{4.1.1}$$

where the a's and b's are the coefficients and the $x_i, i = 1...d$ represent the d relational selectivities.

For ease of exposition, from here onwards we will assume $2D$ selectivity space, where the variables x and y denote the selectivity along the two dimensions. The cost function for $2D$ selectivity space is defined as,

$$Cost(x, y) = a_0 + a_1x + a_2y + a_3xy + a_4x\log x + a_5y\log y + a_6xy\log xy \tag{4.1.2}$$

4.1.1 Cost value interpolation with Regression

We estimate the cost values for the un-optimized points in the following steps.

1. For each plan, say p_k among the α_s plans discovered by s optimizations,
   (a) Extract the cost values associated with the optimized points occupied by p_k.
   (b) Fit the function defined in Equation 4.1.2 with these data points (i.e. selectivities and costs) using linear least square regression method. Let us call the cost function of p_k as $cost_k(x, y)$.
2. For each un-optimized point q_{in} inferred with the plan p_k, evaluate $c_A(q_{in})$ by providing selectivities of q_{in} to the function $cost_k(x_{in}, y_{in})$.

4.1.2 Approximation Errors

The simple approach described above is not enough to generate a high quality cost diagram because there are some side-effects, discussed later, associated with the least square regression technique. We address some of these problems inherent to the regression method and propose solutions to counteract them.

4.1.2.1 Scarcity of Training Data

One of the problems encountered is the unavailability of adequate number of optimized points for fitting the model for each plan, present in the approximate plan diagram. This ocurrs due to plan skew where most of the plans occupy very few query points in the true diagram and even fewer optimized points in the approximate diagram. For example if a plan appears in less than 7 optimized points in a $2D$ approximate plan diagram, the quality of the function (according to Equation 4.1.2) fitted through regression becomes questionable.

Solution: Our experience has been that all the terms $(a_0, a_1, \ldots, a_6)$ present in the cost model may not be necessary to express each and every plan cost function e.g. we can

ignore the "log" terms in absence of *Sort* operator in the respective plan tree. Therefore one simple approach to address this issue would be to throw or keep terms according to their contribution in defining the cost model. This relates to a well known area of research known as *Perturbation Analysis* [44]. Perturbation analysis asks how some quantity will change as a result of changes in one or more underlying parameters. We explore a game theoretic approach for performing the perturbation analysis of our cost model. Before moving ahead, we provide a brief background of *Coalitional Games* and *Shapley Value* which will be used later to illustrate the solution procedure.

Coalitional Games: A coalitional model [59] focuses on what groups of players can achieve rather than on what individual players can do. We now describe a simple version of a coalitional game, namely a coalitional game with *transferable payoff* [59] (*transferable payoff means that there is no restriction on how the total payoff may be divided among the members of a group*).

Coalitional Games with transferable payoff: Formally, the game is a finite set of players G, called the *grand coalition* and a *characteristic function* $v : 2^G \rightarrow \Re^{\geq 0}$. v has the following properties,

1. $v(\Phi) = 0$
2. $v(S \cup T) \geq v(S) + v(T),\ whenever\ S \cap T = \Phi$

The interpretation of the function v is as follows: if S is a coalition of players which agree to cooperate, then $v(S)$ describes the total expected gain from this cooperation, independent of what the players outside of S do. The additivity condition (second property) expresses the fact that collaboration can only help but never hurt.

Shapley Value: In game theory, a *Shapley value* describes one approach to the fair allocation of gains obtained by cooperation among several players.

The Shapley value is one way to distribute the total gains to the players, assuming that they all collaborate. The amount a player i gets which is equal to his/her expected

marginal contribution to all possible sub-coalition is defined as,

$$\phi_i(v) = \sum_{S \subseteq G \setminus \{i\}} \frac{|S|!\,(\omega - |S| - 1)!}{\omega!}(v(S \cup \{i\}) - v(S)) \tag{4.1.3}$$

where $\omega = |G|$ is the total number of players and the sum extends over all subsets S of G not containing player i. The formula can be justified if one imagines the coalition being formed one player at a time, with each player demanding their contribution $v(S \cap i)v(S)$ as a fair compensation, and then averaging over the possible different permutations in which the coalition can be formed.

Mapping our Problem to Game theory: In our case the players participating in the coalitional game are the individual terms $(a_0, a_1 \ldots a_6)$ participating in defining the cost model. We are trying to find out contribution of each such term through computing the Shapley value. To do the same, we need to eliminate the terms and redo the model fitting iteratively. Now as the characteristic function we use the *Goodness of Fit* [72] factor of the cost model perturbed through removing the associated parameters.

Goodness of Fit: The *Goodness of Fit* of a statistical model describes how well it fits a set of observations. Measures of *Goodness of Fit* typically summarize the discrepancy between observed values and the values expected under the model in question. One of the popular *Goodness of Fit* test is the **Coefficient of Determination**(R^2) Test [72], which represents the proportion of variability in the observed(Y_i) and expected$(\hat{Y}_i)$ data. It requires measuring the *Sum of Squares due to Error (SSE)* of the data (Y_i), which is $\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$ and *Total Sum of Squares (SST)*, which is $\sum_{i=1}^{n}(Y_i - \bar{Y})^2$, where $\bar{Y}$ is mean of the observed data. The final value of R^2 is determined as,

$$R^2 = 1 - \frac{SSE}{SST} \tag{4.1.4}$$

One obvious question that might arise is "*Why not use the RMS Error instead of* R^2 *?*". The reason is that *RMSE* value is a good representative of *Goodness of Fit*, given that the model is estimated over the entire population, not just on the samples drawn

from the population. In our case we determine the R^2 value on the set of sample points.

This method of perturbing the cost model can be closely related to the stream of research known as *Multi-perturbation Shapley value Analysis* (MSA)[16, 53]. MSA evaluates the functional contributions by viewing a set of multi-perturbation experiments as a coalitional game as we do. In MSA the desired set of contributions is calculated by the Shapley value too, capturing the unique fair division of the each of function-terms performance. The higher an elements contribution according to the Shapley value, the larger is the part it causally plays in the successful performance of the function.

Algorithm 4 Cost model dimensionality reduction

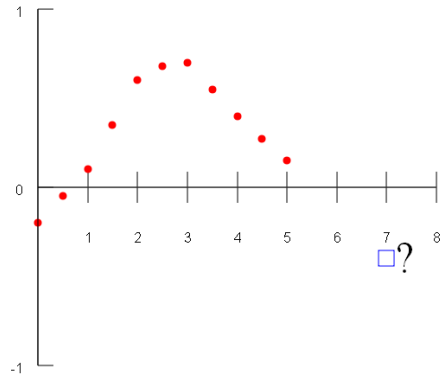
ModelForLowDim(NumCoeffs ω, Plan p_k,CostFunction $costFunc_k$)

1: Generate all possible strings from 0 and 1 of size ω.
2: Set a mapping between each function term to each position in the string.
3: **for** each term a_k, $k = 1$ to ω **do**
4: **for** each such string from $i = 1$ to 2^ω such that the k^{th} position is 0 **do**
5: Build a function $cost_i$ consisting of terms whose corresponding positions are 1 in the string i. {we can reuse this value for subsequent iteration}
6: Curve fit with $cost_i$ (for plan p_k).
7: Evaluate ${R^2}_i$ for $cost_k$
8: Curve fit with $cost_{ki} = cost_i \cup k_{th}\ term$ (for plan p_k).
9: Evaluate ${R^2}_{ki}$ for $cost_{ki}$
10: Calculate value $\phi_{ki} = \frac{\omega_i!(\omega-\omega_i-1)!}{\omega!}\left({R^2}_{ki} - {R^2}_i\right)$, where ω_i is the no. of terms in $cost_i$.
11: **end for**
12: Calculate Shapley value ϕ_k as $\sum \phi_{ki}$.
13: **end for**
14: **return** $s(p_k)$ terms with highest Shapley value.

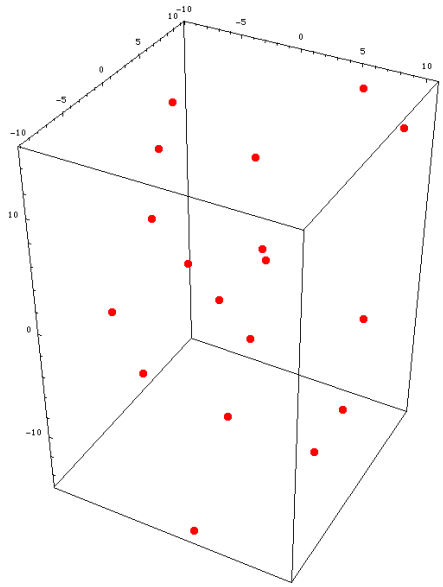
Our method works by starting with a full model and then starts dropping $1, 2 \ldots \omega$ terms one at a time. After Shapley value is calculated for each of the terms, we keep the top $s(p_k)$ terms where $s(p_k)$ is the number of optimized points available for fitting.

4.1.2.2 Effect of Extrapolation

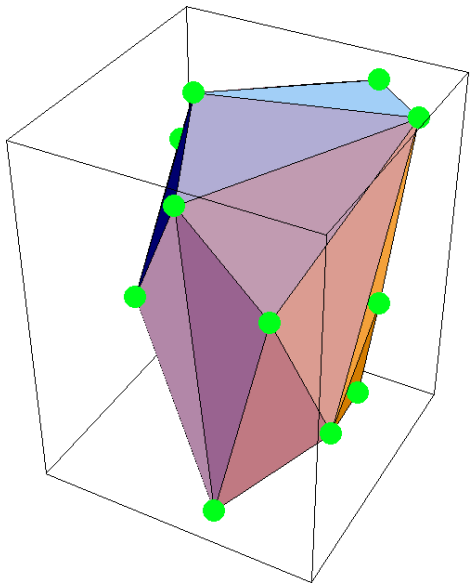
Whenever a linear regression model is fitted to a group of data, the range of the data should be carefully observed. Prediction outside the range of the data used to construct the model is known as extrapolation (refer to Figure 4.1(a)) and it is risky. The range of



(a) **Extrapolation (Source: *http://en.wikipedia.org/wiki/Extrapolation*)**



(b) **Set of optimized points**



(c) **Convex Polytope formed by the points**

Figure 4.1: Convex Polytope (Source: *http://xcellerator.info/mPower/pages/convexHull.html*)

the data deciding cost function of a plan can be described as the ***Convex Polytope*** (refer to Figure 4.1) formed in A. Therefore, inferring cost value with the same function at a query point lying outside the convex polytope of the plan can be treated as extrapolation in our scenario. Introduction of this error is highly probable in our approach, as at the time of inferring the un-optimized points we inflate the area of an individual plan than that is covered by the optimized points in A.

Solution: We can first extract the convex polytope (CP_k for plan p_k) formed by the optimized points in A and refrain from using the cost model developed for the interior points to estimate cost for the exterior points. However, algorithms designed for extracting convex polytopes are computationally expensive and their high dimensional counterparts are not so popularly known in practice. Fortunately we can bypass identification of convex polytope to tell if a point q lies inside CP_k by checking whether or not q can be expressed as convex combination of all the optimized points inside CP_k. This can be obtained efficiently by checking if feasible solution to the standard form linear programming (LP) problem with degenerate objective function exists or not. The function is of the form $Ax = b$, where A is a $[d \times n_k]$ matrix, n_k being the number of points lying inside CP_k in a d dimensional selectivity space. b is $[d \times 1]$ matrix representing the coordinates of the point q. Therefore solution for x which is a $[n_k \times 1]$ matrix, will give the desired linear combination. All we have to check is if $\forall_i, 0 \leq x_i \leq 1$ and $\sum_{i=1}^{n_k} x_i = 1$, where x_i is the i^{th} entry of the solution matrix x.

However, it would become tremendously expensive if we intend to check this for all the un-optimized query points in selectivity space, which can be around 90% if approximation with GS_PQO algorithm is employed. Fortunately, we can reduce the effort by considering only those points which are assigned with a plan covering $\leq 50\%$ optimized points at the nearest neighbor. For a 90% accurate diagram this reduces the target points to roughly around 10%(empirically verified). We check the LP solution for these points and if they actually lie exterior to the plan polytope then we apply linear extrapolation technique to infer the cost value instead of using the cost model defined for the points inside the polytope. Note that, since we only consider optimized points to construct the matrix A, n_k is limited to a small value and hence the matrix solution does not impose significant overheads.

The outline of this method is described in Algorithm 5.

One can raise a question about the choice of linear extrapolation over polynomial or other popular nonlinear techniques. However, in our case as the extrapolation bandwidth i.e. the plan boundary is limited to a reasonably small value, we find linear extrapolation

Algorithm 5 Extrapolating cost model

TreatExtrapolationNOutliers(SetofPoints Q)

for all un-optimized point q_i such that $0 \leq i < n$, from the set Q **do**
  if q_i is inferred by a plan occupying $\leq 50\%$ of optimized points **then**
    set $eFlag[q_i]$ = **true**
  else
    set $eFlag[q_i]$ = **false**
  end if
end for
for all inferred point q_i such that $0 \leq i < n$, from the set Q **do**
  if $eFlag[q_i ==$ **true then**
    Find solution of $Ax = b$.
    if $[\forall_i, 0 \leq x_i \leq 1$ and $\sum_{i=1}^{n_k} x_i = 1]$ is violated **then**
      find two nearest optimized points q_1 and q_2 in the direction of the center of the polytope.
      if two such points are found **then**
        set $c(q_{in}) = c(q_1) + \frac{dist(q_{in} - q_1)}{dist(q_2 - q_1)} \times (c(q_2) - c(q_1))$
      else if only one such point found **then**
        apply linear scaling to determine the cost using distance from origin.
      end if
    end if
  end if
end for

sufficient for our purpose. Note that for finding out the matrix solution we can not use standard ***LU decomposition*** scheme as the matrix in this case may not be square. We apply ***QR decomposition*** scheme to obtain the same. The implementation details are given in Section 6.

4.2 Class II optimizer

For Class II optimizers with FPC, the exact cost diagram respective to the approximate plan diagram can be obtained without doing the regression and interpolation steps mentioned in the previous section. An additional overhead is incurred due to this e.g. if there are 90% inferred points in the diagram hence performing FPC on all of them would add an overhead of 9% (considering one FPC operation takes $\frac{1}{10}^{th}$ of the optimization). The errors incurred in Class II cost diagrams obtained through FPC are due to location error only, where costing a sub-optimal plan can give rise to spikes in the cost diagram.

However, location error incurred in Class II is low as we use FPC to resolve ties between different plans contending at an un-optimized points.

Alternative approach for Class II. For Class II optimizers we can perform a better job of cost inference without explicitly executing FPC on every query point. We can perform FPC only at points which suffer from extrapolation or curse of dimensionality error. Empirically we have seen these reduces around 50% of the total FPC cost keeping the quality of final cost diagram comparable to the earlier approach.

4.3 Class III optimizer

The Class III algorithms i.e. *PlanFill* and *Relaxed-PlanFill* both operate by costing the entire query space at least once, therefore cost diagram is a byproduct of the algorithm itself. For *PlanFill* we obtain the zero error approximation of cost diagram. Whereas *Relaxed-PlanFill* admits error in cost diagram approximation at the points suffering location error similar to the Class II diagrams.

4.4 Experimental Results

We use the same experimental setup as the one described in Section 3.5, for plan diagram approximation experiments. In the remainder of this section we present performance of cost diagram approximation. Note that we do not optimize any further query points during this process. These error measurements are different from the results shown in Table 3.14 which showed how much damage one can suffer by choosing the sub-optimal plans from the erroneous locations of A. This section strictly speaks about efficiency of the cost diagram approximation techniques. So another aspect of approximation efficiency in this case relies on how well the cost of a plan outside its optimality region can be estimated.

The performance of cost approximation technique is presented in Table 4.1. To measure the percentage error at a point, we evaluate the difference between the actual cost of the plan assigned at that point and the estimated value both taken strictly from A. For

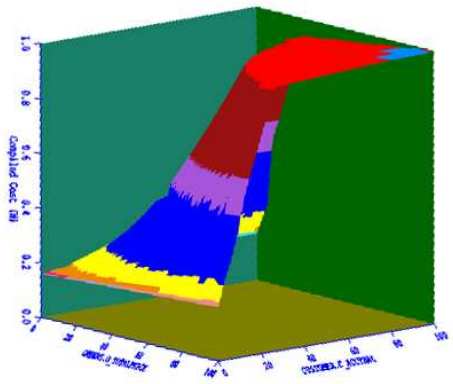
points suffering location error we determine the actual cost by performing FPC (available with OptCom) at that point feeding the inferred plan as an input. We showcase results produced only by GS_PQO. The approximation errors will be prevalent in GS_PQO as it involves lesser number of optimizations.

We have shown results for both the naive and corrected interpolation techniques. It is evident that in terms of quality we gained a lot with the corrected version of interpolation technique. Specifically for $QT8$, the 100-3D diagram, the error(ϵ_{max} was improved from 100% to 25%, similarly for $DSQT25b$ there is a drastic improvement from 9289.58% to 0.65%. Note that, we did not allow assignment of negative cost values in any of the techniques mentioned. In both the cases if the model estimates negative value, we ignore that value and assign cost of the nearest neighbor.

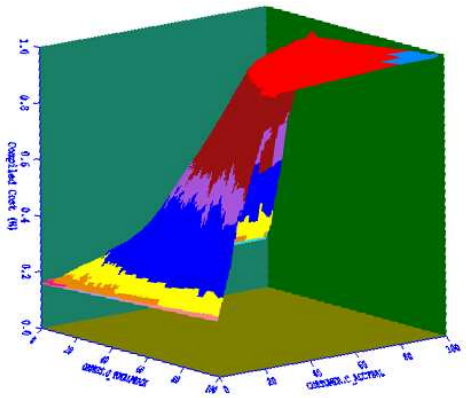
Figure 4.2 demonstrates the approximation quality for some of the complex diagrams encountered by us. The shape of the diagrams are retained well after approximation, though the existence of unevenness in the diagram is mainly due to presence of small plans there. As an example in Figure 4.2(f), the small spikes in the cost diagram is due to introduction of location error around those region of the plan diagram.

Dimension /Resolution	Query Template	W/O Correction		With Correction	
		ϵ_{max} (%)	ϵ_{RMS}(%)	ϵ_{max} (%)	ϵ_{RMS}(%)
2D:100X100	QT2	26.35 %	2.40 %	26.35 %	2.40 %
	QT3	68.87 %	6.57 %	5.65 %	1.10 %
	QT4	3.14 %	0.29 %	3.14 %	0.29 %
	QT5	20.02 %	1.22 %	6.71 %	1.19 %
	QT7	19.50 %	2.20 %	11.91 %	1.36 %
	QT8	74.24 %	2.06 %	2.17 %	0.40 %
	QT9	71.23 %	11.30 %	4.06 %	0.19 %
	QT10	83.70 %	1.19 %	6.07 %	0.13 %
	QT11	50.80 %	17.66 %	0.55 %	0.04 %
	QT16	7.82 %	0.47 %	4.24 %	0.45 %
	QT17	5.61 %	0.12 %	0.32 %	0.06 %
	QT18	4.90 %	0.21 %	0.81 %	0.07 %
	QT20	86.30 %	8.03 %	0.37 %	0.06 %
	QT21	9.93 %	1.45 %	9.93 %	1.47 %
2D:300X300	QT2	82.00 %	9.36 %	42.30 %	3.20 %
	QT3	65.56 %	1.17 %	4.74 %	0.96 %
	QT4	7.28 %	0.41 %	7.28 %	0.41 %
	QT5	53.00 %	41.63 %	8.12 %	1.21 %
	QT7	36.10 %	1.72 %	22.39 %	1.68 %
	QT8	70.65 %	1.31 %	1.75 %	0.19 %
	QT9	72.73 %	1.32 %	1.32 %	0.27 %
	QT10	115.00 %	13.07 %	46.00 %	3.10 %
	QT11	0.58 %	0.05 %	0.58 %	0.05 %
	QT16	6.61 %	0.53 %	4.29 %	0.49 %
	QT17	0.48 %	0.06 %	0.48 %	0.06 %
	QT18	6.35 %	0.30 %	5.43 %	0.29 %
	QT20	42.00 %	6.40 %	12.90 %	2.10 %
	QT21	13.06 %	1.39 %	13.06 %	1.35 %
2D:1000X1000	QT8	69.35 %	0.48 %	1.85 %	0.40 %
	QT16	92 %	8.50 %	6.69 %	0.10 %
	QT21	25.44 %	1.49 %	16.86 %	1.57 %
3D:100X100X100	QT8	100 %	22.00 %	25.83 %	0.37 %
	QT9	49 %	18.90 %	16.92 %	0.37 %
	QT21	72.96 %	10.15 %	13.09 %	2.89 %
2D:100X100	DSQT2	0.28 %	0.03 %	0.28 %	0.03 %
	DSQT17	78.26 %	4.20 %	1.98 %	0.05 %
	DSQT18	67.70 %	5.80 %	7.78 %	0.76 %
	DSQT19	67.35 %	4.31 %	9.00 %	0.66 %
	DSQT25	94.31 %	2.78 %	7.91 %	0.25 %
	DSQT25a	2.16 %	0.15 %	2.16 %	0.16 %
	DSQT25b	9289.58 %	109.80 %	0.65 %	0.06 %
	DSQT50	3.90 %	1.04 %	3.90 %	1.04 %
	DSQT76	71.21 %	6.47 %	35.00 %	1.50 %
2D:300X300	DSQT12	17.02 %	8.24 %	11.14 %	3.93 %
	DSQT18	48.46 %	6.63 %	17.14 %	0.95 %
	DSQT19	50.00 %	37.90 %	12.76 %	0.69 %
	DSQT28	15.94 %	0.48 %	14.92 %	0.36 %
3D:100X100X100	DSQT19	73.52 %	1.73 %	15.47 %	0.72 %

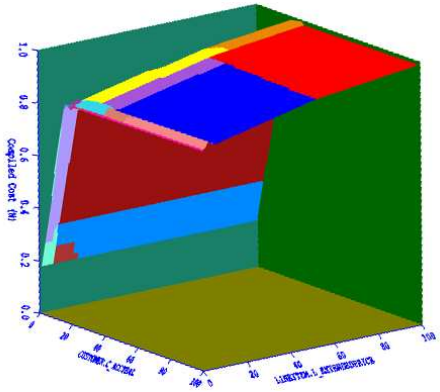
Table 4.1: Cost approximation error for 90% accurate plan diagram



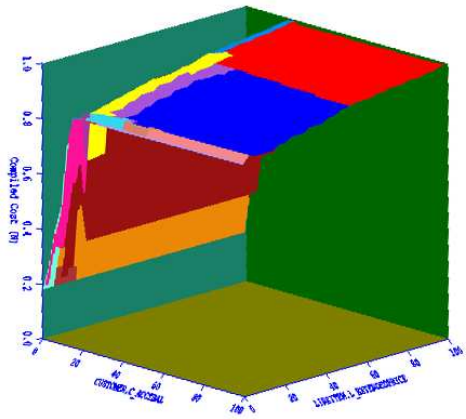
(a) **Original Cost Diagram : QT7 (TPC-H)**



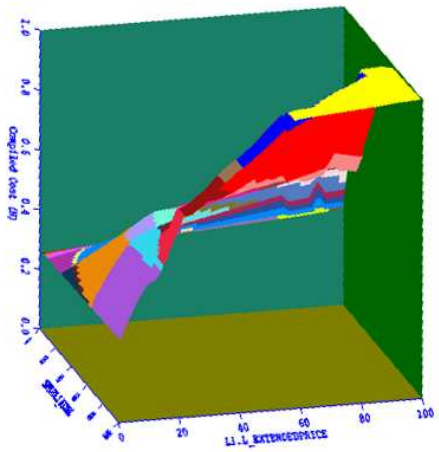
(b) **Approx Cost Diagram : QT7 (TPC-H)**



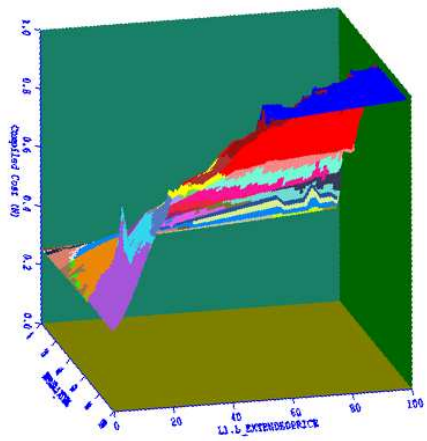
(c) **Original Cost Diagram : QT10 (TPC-H)**



(d) **Approx Cost Diagram : QT10 (TPC-H)**



(e) **Original Cost Diagram : QT21 (TPC-H)**



(f) **Approx Cost Diagram : QT21 (TPC-H)**

Figure 4.2: Qualitative Performance of Cost Approximation

Chapter 5

Cardinality Diagram Approximation

Having considered plan and cost diagrams, in this chapter we move on to discussing techniques to approximate the last type of *Optimizer Diagrams* namely cardinality diagrams.

We begin by designing a parametrized mathematical model for characterizing plan cardinality behavior using a similar approach to that made for cost presented in [19], which was discussed in the previous chapter. The cardinality estimation techniques adopted by different database query optimizers are mostly implemented as an extension to the pioneering work presented in [68]. Our model also tries to capture the same. We have found in practice that with appropriate settings of the parameters our model is quite accurate with many distinct cardinality diagrams arising out of TPC-H and TPC-DS-based query templates on industrial optimizers, both behaviorally and quantitatively.

Let us consider the true cardinality of a query point $q(x, y)$ in P is $C_P(q)$ and with our approximation technique the cardinality value is estimated as $C_A(q)$ in A. We denote $\max(C_P) = \max_{j=1}^{n} C_P(q_j)$ as the maximum cardinality of a query point obtained in P. The quality metrics used are similar to that defined in Chapter 4 for cost approximation. The metrics $\epsilon_{abs}(C_i)$, $\epsilon_{rel}(C_i)$ and $\epsilon(C_i)$ are defined below.

$$\begin{aligned} \epsilon_{abs}(C_i) &= |C_P(q_i) - C_A(q_i)| \\ \epsilon_{rel}(C_i) &= \frac{|C_P(q_i) - C_A(q_i)|}{C_P(q_i)} \\ \epsilon_(C_i) &= \min\{\beta_{abs} \times \epsilon_{abs}(C_i), \beta_{rel} \times \epsilon_{rel}(C_i)\} \end{aligned}$$

Similar to cost approximation, we use $\beta_{abs} = 1, \beta_{rel} = 100$. ϵ_{MAX} and ϵ_{RMS} are calculated as:

$$\begin{aligned} \epsilon_{MAX}(\%) &= \max_{i=1}^{n} \epsilon_(C_i) \\ \epsilon_{RMS}(\%) &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\epsilon_{abs}(C_i)}{\max(C_P)}\right)^2} \end{aligned}$$

5.1 Query Cardinality Model

We start by deriving the final output cardinality according to the contribution made by different kinds of nodes in a plan tree. We then present the final cardinality model.

5.1.1 Cardinality Derivation

Let us study the plan operator tree generated by a $2D$ query template shown in Figure 5.1. In current optimizers, the operators in the execution plan tree are all typically either *unary* or *binary* with regard to their inputs. Cardinality on each relational operator or node is derived bottom up. There are only three kinds of *Dependent Nodes*([19]) affecting the output cardinality. The node types are shown in Figure 5.1 and described below.

1. **Filter Node**: Unary nodes with the range or equality predicates on scalar-valued attributes are categorized as Filter Nodes. Note that Filter Node on base relations are the Selectivity Nodes defined in [19]. There may be other filter nodes present at the upper part of a plan tree also for example arising due to presence of HAVING clause. The *blue* nodes in Figure 5.1 denote the filter nodes.

2. **Join Node**: These are binary nodes that host the join clauses between two relations. In Figure 5.1 the *red* colored nodes are join nodes.

3. **Aggregate Node**: These unary nodes perform aggregation (COUNT, SUM, AVG, MAX etc) of the results according to the specifications declared with the "GROUP BY" clause. For example the *green* nodes in Figure 5.1, are aggregate nodes.

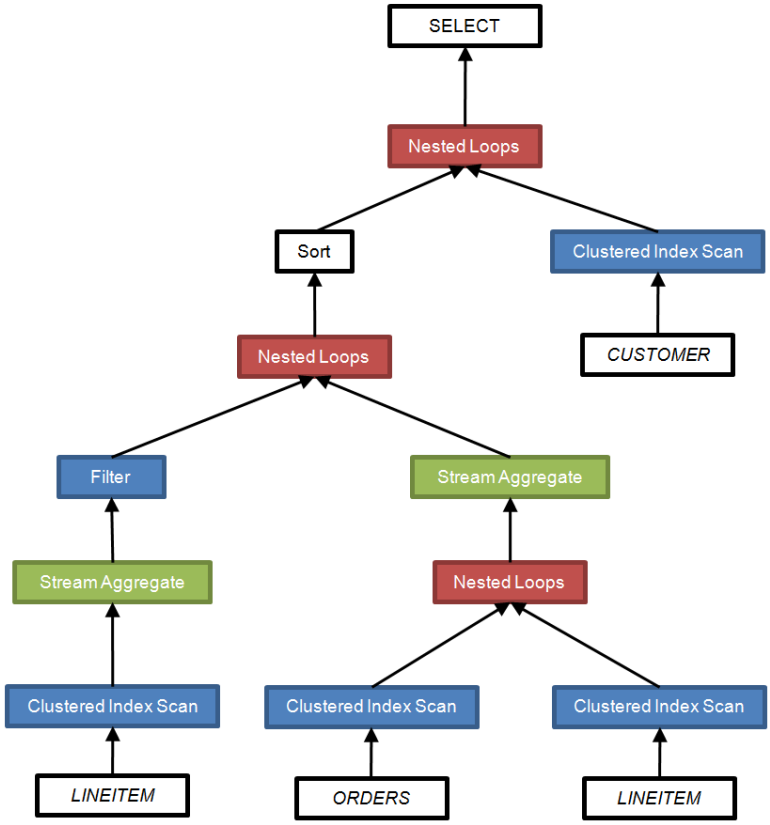


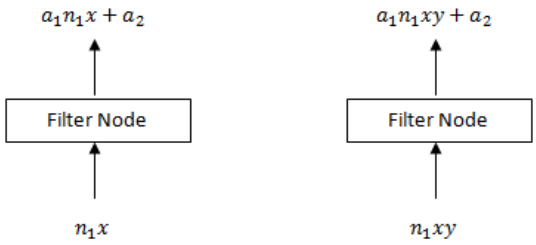
Figure 5.1: Different predicates in a Plan Tree (TPCH - Query Template 18)

5.1.2 Node Cardinality Model

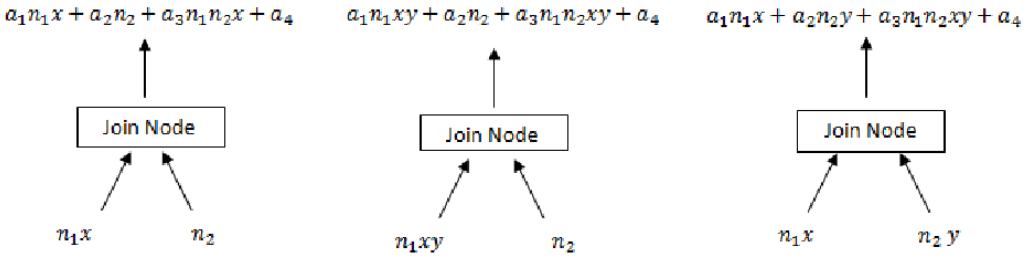
For ease of presentation, we will initially assume that our objective is to model the behavior of the cardinality diagram with respect to a 2D selectivity space. We will assume variable x and y denote the selectivities on respective dimensions and R_x and R_y as the relations contributing "Selectivity Predicates". Let us consider A_x and A_y are the attributes involved.

Note that x and y are obtained after going through the initial Filter Node on the base relations. We derive the model as below,

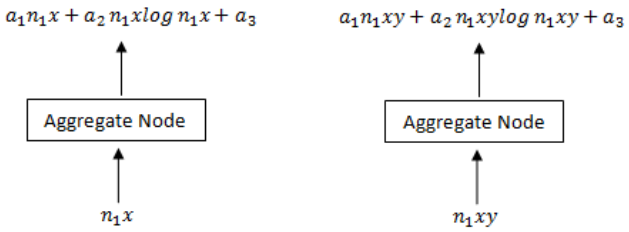
1. **Filter Node:** As mentioned earlier, Filter nodes on base relations are nothing but Selectivity Nodes, whose cardinality model can be expressed as a linear function of x,y and xy as shown in Figure 5.2(a). However, presence of filter above the



(a) **Filter Nodes**



(b) **Join Nodes**



(c) **Aggregate Nodes**

Figure 5.2: Cardinality Model of *Dependent Nodes*

Selectivity Nodes can produce a clamping effect on the final output cardinality. We will discuss how we modeled this later in this section.

2. **Join Node:** With the similar arguments as given in [19] we can model the output cardinality of a Join Node as shown in Figure 5.2(b) with $a_1, \ldots a_4$ being coefficients.

3. **Aggregate Node:** In the Aggregate Nodes, if there is no grouping column then output cardinality of the aggregate node is 1. Otherwise the final value is then clamped according to the maximum possible output row count. Note that clamping

of output for this node can occur only around the end part of the cardinality diagram. We use logarithmic model to capture this nature of Aggregate Nodes. The model associated with aggregate node is shown in Figure 5.2(c).

Modeling the Flat regions. We first start by restricting ourselves only to *monotonic* cardinality diagrams. These means that the we will concentrate on flat regions present either at the lowest or highest selectivity regions. Now the log terms introduced is capable to imitate the flat regions with limited spread (around 25% of the diagram). Therefore, we refine our final cardinality model by introducing two additional expressions which enable us to represent a cardinality diagram with one or more extended flat regions. The idea is to first check the quality of model, only if it is not sufficient we use these extra terms to represent a complete cardinality model.

5.1.3 Complete Cardinality Model

The final output cardinality of a query point can be expressed as the aggregate sum of the cardinality values of the individual nodes. As mentioned earlier to capture the flat regions in the cardinality model which the logarithmic terms can not, we introduce two extra terms. The part related to C_{min} and C_{max} takes care of the extended flat regions present in the cardinality diagram and rest of the points are modeled by the second part of the function. The final model for 2D diagram looks like,

$$Card(x,y) = \begin{cases} C_{min} & \text{if } x \leq x_{min} \text{ and } y \leq y_{min} \\ C_{max} & \text{if } x \geq x_{max} \text{ and } y \geq y_{max} \\ (a_1x + a_2y + a_3xy + a_4xlog(x) + a_5ylog(y) & \text{otherwise} \\ +a_6xylog(xy) + a_7) & \end{cases} \tag{5.1.1}$$

Extension to d-dimension is also straight forward,

$$Card(x_1, ..., x_d) = \begin{cases} C_{min} & \text{if } x_i \leq x_i^{(min)} \\ C_{max} & \text{if } x_i \geq x_i^{(max)} \\ \sum_{i_1}(a_{i_1}x_{i_1} + b_{i_1}x_{i_1}\log x_{i_1}) & \text{otherwise} \\ + \sum_{i_1<i_2}(a_{i_1i_2}x_{i_1}x_{i_2} + b_{i_1i_2}x_{i_1}x_{i_2}\log x_{i_1}x_{i_2}) & \\ + \ldots + a_{12\ldots d}(x_1x_2x_3\ldots x_d) & \\ + b_{12\ldots d}(x_1x_2x_3\ldots x_d)\log(x_1x_2x_3\ldots x_d) + a_0 & \end{cases} \tag{5.1.2}$$

where $i = 1, 2 \ldots d$. From here onwards we will refer the C_{min} and C_{max} related terms as *first part* of the model and the remaining terms as *second part*.

5.2 Cardinality value interpolation with Regression

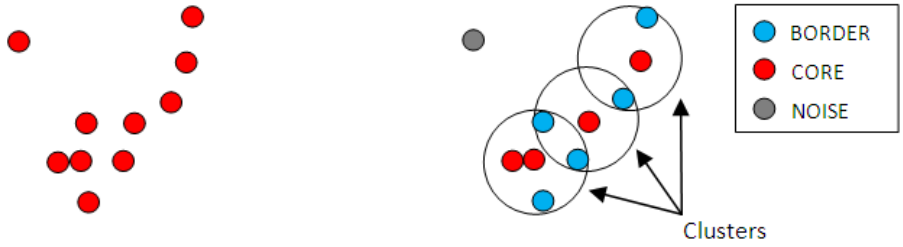
Following similar approach as mentioned in Section 4.1.1 we estimate the cardinality values for the un-optimized points in the following steps.

1. Extract the cardinality values associated with the optimized points.
2. Fit them into the function defined in Equation 5.1.1. Let us call the cardinality function of as $card(x, y)$.
3. For each un-optimized point q_{in}, estimate $C_A(q_{in})$ by feeding selectivities of q_{in} to the function $card(x_{in}, y_{in})$.

However the major differences from the cost interpolation technique are first a) we fit the model for all the optimized points in the diagram i.e. we derive one cardinality model for a diagram and secondly b)the process of fitting the cardinality model to data can not be done entirely through linear least square regression technique. We use DBSCAN clustering mechanism to determine the values related to C_{min} and C_{max} as described in the next section.

5.3 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

DBSCAN algorithm is used later in this chapter for determining parameters related to the *first part* of the cardinality model. Here we briefly describe working of this algorithm and its relation to cardinality diagram approximation.



(a) **Set of data points** (b) **Final Clusters**

Figure 5.3: Example of clustering by DBSCAN

The algorithm DBSCAN, based on the formal notion of density-reachability for d-dimensional points, is designed to discover clusters of arbitrary shape. The runtime of the algorithm is of the order $O(nlogn)$ if region queries are efficiently supported by spatial index structures (such as R* tree), i.e. at least in moderately dimensional spaces. The brute-force approach is of order $O(n^2)$. The key idea is that for each point of a cluster the neighborhood of a given radius has to contain at least a minimum number of points, i.e. the density in the neighborhood has to exceed some threshold. The shape of a neighborhood is determined by the choice of a distance function for two points p and q, denoted by $dist(p, q)$. For instance, when using the Manhattan distance in 2D space, the shape of the neighborhood is rectangular. DBSCAN requires two parameters: cluster radius (l) and minimum points (m). It starts with an arbitrary starting point that has not been visited. It then finds all the neighboring points within distance l of the starting point. If the number of neighbors is greater than or equal to m, a cluster is formed.

The starting point and its neighbors are added to this cluster and the starting point is marked as visited. The algorithm then repeats the evaluation process for all the neighbors recursively. If the number of neighbors is less than m, the point is marked as noise. If a cluster is fully expanded (all points within reach are visited) then the algorithm proceeds to iterate through the remaining unvisited points in the data set. A graphical illustration of DBSCAN is shown in Figure 5.3.

The outline of DBSCAN algorithm is given below:

Algorithm 6 DBSCAN

DBSCAN(ClusterRadius l,MinSize m,Data D,Cluster Φ,ClusterNo Φ_{last})

for each unvisited point q_u in data set D **do**
  $N \leftarrow getNeighbors(q_u, l)$
  if $sizeof(N) < m$ **then**
    Mark q_u as NOISE
  else
    $++\Phi_{last}$.
    Mark q_u as VISITED.
    Add q_u to cluster Φ.
    DBSCAN(l,m,D,ϕ,Φ_{last}).
  end if
end for

5.3.1 Interpretation of DBSCAN in our setup

We use DBSCAN to retrieve clusters consisting of query points with very close cardinality values and are continuous in selectivity region, hence both the *BORDER* and *CORE* points belong to a cluster in our case. In this process the query points categorized as *NOISE* are the uninteresting points i.e. they will be fitted through the *second part* of the cardinality model defined in Equation 5.1.2 through the least square regression technique. Note that all the points dealt with are genuine points and these terms have nothing to do with the quality of individual cardinality values. The related parameters are set as described below.

1. The feature set contains two attributes, $cardinality, selectivity$.

2. The initial values are set to $l = 0.001$ and $m = 0.05 \times n$.

3. The distance between two neighboring points here means the absolute difference in their cardinality values and the chessboard distance in selectivities.

5.4 Class I Optimizers

Similar to cost diagram approximation, we consider the approximate plan diagram as an input for cardinality approximation. But unlike cost, cardinality function is determined for the entire plan diagram rather than for individual plans. The cardinality model derived above involves 13 parameters in $2D$ selectivity space.

5.4.1 Approximation techniques

Our approximation technique starts by fitting the *second part* of cardinality model (as defined in Equation 5.1.2) with the set of optimized data points obtained after plan diagram approximation. We use linear least square regression technique for this purpose. After which we calculate the *Goodness of Fit* of the regression with R^2 test defined in Equation 4.1.4 in the previous chapter. If the R^2 value evaluates to be ≥ 0.6, we ignore first part of the model. Otherwise we apply DBSCAN to extract the cluster corresponding to C_{min} and C_{max}. Finally after the model is derived, it is used to determine cardinality value for each inferred point. We do not need to apply DBSCAN unless it is absolutely necessary. The outline of the algorithm(7) is given below (assume 2D model),

Note that the approximation errors mentioned for cost diagram approximation do not play a major role here since (a) we construct the model for the entire diagram, the number of optimize points are adequate enough for a good quality fitting, and (b) we optimize the corner points for initial grid processing, which ensures all the inferred points are within same convex polytop.

Algorithm 7 Approximating cardinality diagram

1: $x_{min} \leftarrow y_{min} \leftarrow 0$; $x_{max} \leftarrow y_{max} \leftarrow r$;
2: Fit data to the *second part* of cardinality model defined in Equation 5.1.1 using linear least square regression technique.
3: Calculate R^2;
4: **if** $R^2 < 0.6$ **then**
5: $\Phi \leftarrow \emptyset; \Phi_{last} \leftarrow 0$.
6: Call $DBScan(l, m, \Phi, \Phi_{last})$.
7: Extract clusters $\hat{\Phi}$ which is associated with minimum and maximum cardinality values C_{min} and C_{max} respectively.
8: Run a single scan through the data points of $\hat{\Phi}$ to determine x_{min}, y_{min} and x_{max}, y_{max}.
9: Fit data $\notin \hat{\Phi}$ to the *second part* of cardinality model defined in Equation 5.1.1.
10: **end if**
11: **return** final model.

5.5 Class II

For Class II using FPC feature we can determine the true cardinality of the inferred points. This is because unlike the cost diagram, location error does not affect the output cardinalities of inferred points since cardinality is not a function of plans assigned there. Therefore the above mentioned techniques are not quite required for Class II optimizers unless one intends to save the number of FPC performed as mentioned in Section 4.2 in the previous chapter.

5.6 Class III

Since the plan diagram approximation algorithms perform FPC at each query point at least once, we obtain the zero error cardinality diagram as a bonus output. Therefore, we do not employ the above mentioned procedures of cardinality approximation for Class III optimizers.

5.7 Experimental Results

In this section we will present experimental results to strengthen the claim on quality of the cardinality model and the approximation error caused by it.

5.7.1 Model Quality.

We tried fitting all the query points present in the selectivity space on the cardinality model, with an intention to observe the behavior of the fitted models. As expected, the fitted models retained the shape of the original cardinality diagrams. The quality of the fitted diagram will depict the applicability of the cardinality model. Figures 5.4, 5.5, 5.6 and 5.7 show the shape of some of these true and fitted cardinality diagrams. In Figure 5.4, the absolutely flat cardinality diagram for $QT8$ is correctly modeled, similarly the model derived for the staircase (Figure 5.5) or "L-shaped" (Figure 5.7) cardinality diagrams have retained their original shape too. Among these diagrams only for TPCH QT18 (Figure 5.7) we required both *first* and *second* parts of the cardinality model i.e. executed the DBSCAN algorithm. Note that the color in the plots presented in Figures 5.4, 5.5, 5.6 and 5.7 do not denote plan color, the colors act as a differentiator between cardinality values for better visualization.

Stronger results are obtained when experimented on many distinct diagrams across different optimizers for both TPCH and TPCDS benchmarks. The results are given in the Table 5.1, where we can see that the maximum and average RMS error is less than 15% and 5% respectively i.e. the proposed model came out to be around 95% accurate on an average.

Database	**Dimension**	**# of Templates)**	**Max RMS error (%)**	**Avg RMS error (%)**
TPC-H	2D	310	11.9%	4.1 %
TPC-H	3D	120	8.1%	3.3 %
TPC-DS	2D	140	14%	4.5 %
TPC-DS	3D	50	7.3%	3.2 %

Table 5.1: Quantitative performance of Cardinality Model over different DB-Engines

5.7.2 Approximation Quality.

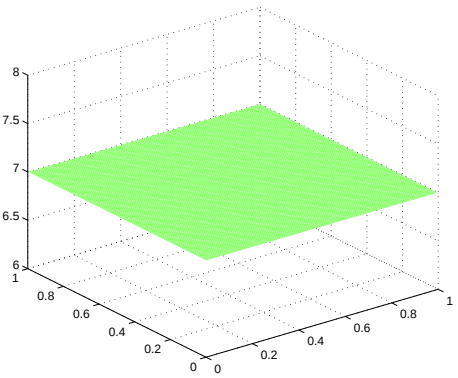
As mentioned earlier, we consider plan diagram to be the input for the cardinality approximation process. First the model is derived through linear least square regression and DBSCAN (if required) performed on the output cardinalities at the optimized query

points. Finally the cardinality function is used to estimate cardinalities for the inferred points. We ran experiments on some of the representative query templates of TPC-H and TPC-DS benchmarks. Table 5.2 lists the maximum RMS error (ϵ_{RMS}) and maximum error (ϵ_{MAX}) incurred by approximating cardinalities for these query templates. The low RMS value indicates that the error across the diagram is small, whereas the presence of steep rise in cardinality introduces significant error for some points. This is because the exact selectivities where these sudden changes occur are very difficult to catch through a continuous model. Nevertheless, our model is well equipped to imitate the clamping nature brought in by GROUP BY or HAVING clauses, hence the overall error i.e. ϵ_{RMS} value evaluates to be very small.

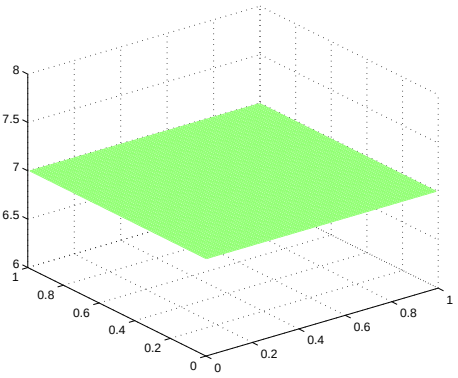
Finally the quality of approximate cardinality model can be observed from Figure 5.4, 5.5, 5.6 and 5.7. These diagrams show that with our approximation technique the original shape could be retained. The spikes seen in the approximate diagrams are due to the presence of optimized points on those locations, where the nearby points are inferred through the fitted model.

Dimension / Resolution	Query Template	ϵ_{MAX} (%)	ϵ_{RMS} (%)
2D:100X100	QT2	19.8 %	6.01 %
	QT3	29.60 %	8.76 %
	QT4	0.00 %	0.00 %
	QT5	5.61 %	1.74 %
	QT7	23.51 %	1.85 %
	QT8	0.00 %	0.00 %
	QT9	59.69 %	11.35 %
	QT10	25.02 %	5.57 %
	QT11	15.54 %	1.28 %
	QT16	19.57 %	3.62 %
	QT17	0.00 %	0.00 %
	QT18	11.72 %	8.03 %
	QT20	23.82 %	3.87 %
	QT21	30.30 %	4.35 %
2D:300X300	QT2	20.25 %	6.24 %
	QT3	31.40 %	8.95 %
	QT4	0.00 %	0.00 %
	QT5	85.05 %	2.93 %
	QT7	53.30 %	2.07 %
	QT8	0.40 %	0.05 %
	QT9	70.00 %	0.50 %
	QT10	29.31 %	7.31 %
	QT11	16.59 %	1.32 %
	QT16	20.90 %	3.85 %
	QT17	0 %	0 %
	QT18	44.07 %	7.13 %
	QT20	28.69 %	4.05 %
	QT21	38.94 %	5.31 %
2D:1000X1000	QT8	78.14 %	0.27 %
	QT16	24.39 %	4.94 %
	QT21	52.21 %	5.69 %
3D:100X100X100	QT8	0 %	0 %
	QT9	75.18%	8.33 %
	QT21	66.93 %	8.1 %
2D:100X100	DSQT2	0.01 %	0.00 %
	DSQT17	19.92 %	1.90 %
	DSQT18	39.65 %	7.12 %
	DSQT19	30.35 %	8.91 %
	DSQT25	23.72 %	5.36 %
	DSQT25a	60.09 %	16.91 %
	DSQT25b	31.13 %	9.61 %
	DSQT50	5.74 %	1.62 %
	DSQT76	37.20 %	3.59 %
2D:300X300	DSQT12	53.89 %	12.68 %
	DSQT18	43.88 %	13.90 %
	DSQT19	31.51 %	8.70 %
	DSQT28	30.44 %	5.86 %
3D:100X100X100	DSQT19	55.81 %	7.43 %

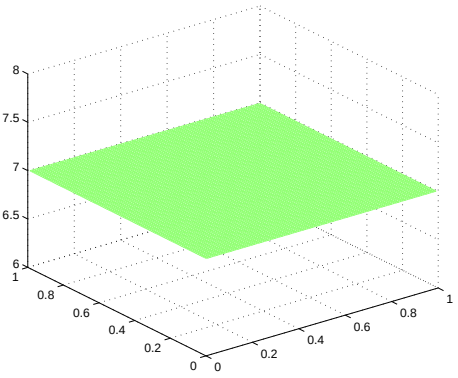
Table 5.2: Maximum Cardinality Error due to Approximation [OptCom]



(a) Original

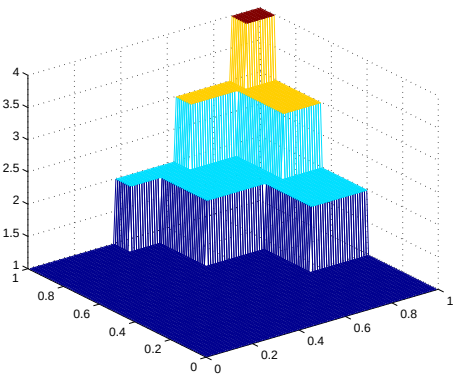


(b) Fitted with Model

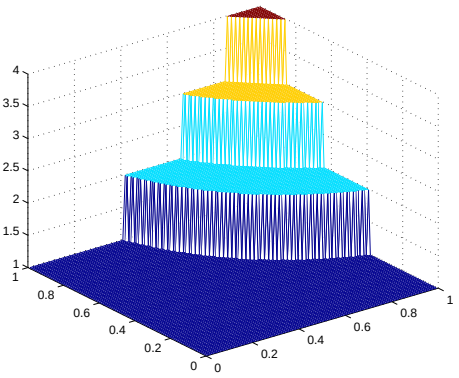


(c) Approximate ($\theta = 10\%$)

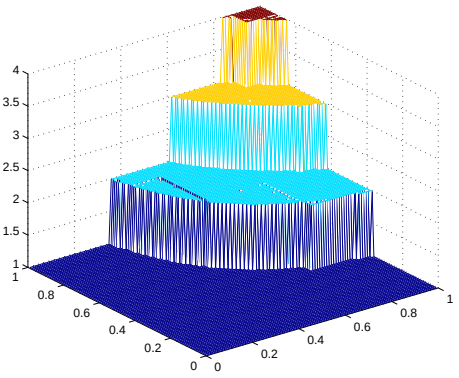
Figure 5.4: Cardinality Diagrams for TPCH QT8



(a) Original

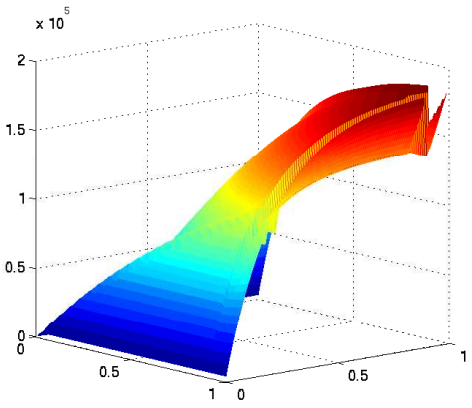


(b) Fitted with Model

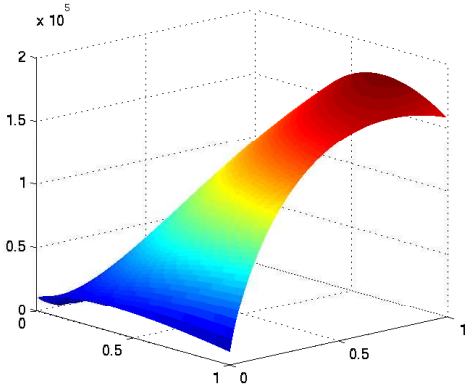


(c) Approximate ($\theta = 10\%$)

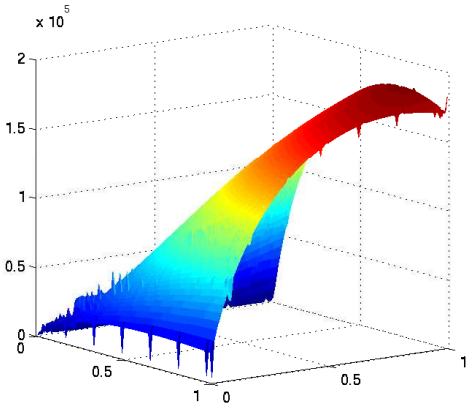
Figure 5.5: Cardinality Diagrams for TPCH QT9



(a) Original

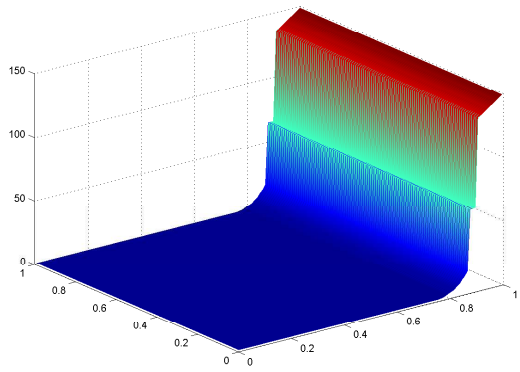


(b) Fitted

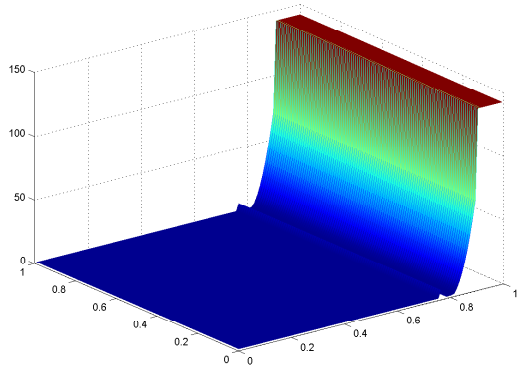


(c) Approx

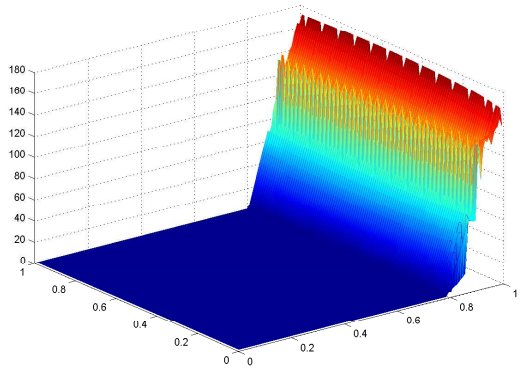
Figure 5.6: Cardinality Diagrams for TPCH QT16



(a) Original



(b) Fitted with Model



(c) Approximate ($\theta = 10\%$)

Figure 5.7: Cardinality Diagrams for TPCH QT18

Chapter 6

Implementation in Picasso and PostgreSQL

The various algorithms proposed in this thesis have been implemented in the publicly available query optimizer visualization tool Picasso v2.0. Additionally we have also modified PostgreSQL v8.3.6 to incorporate the foreign plan costing and plan rank list features.

6.1 Picasso

Picasso has been developed in the Database Systems Lab [74] at the Indian Institute of Science, for visually analyzing the behavior of industrial-strength relational query optimizers. It generates a host of diagrams that throw light on the functioning of the optimizer for a parameterized query template over the relational selectivity space. Given a query template, the grid resolution, the distribution at which the instances of this template should be spread across the selectivity space, the parameterized relations (axes) and their attributes on which the diagrams should be constructed, and the choice of query optimizer, the Picasso tool automatically generates the associated SQL queries, submits them to the optimizer to generate the plans, and finally produces the color-coded plan, cost and cardinality diagrams.

A block diagram of the Picasso architecture is shown in Figure 6.1. Every request from

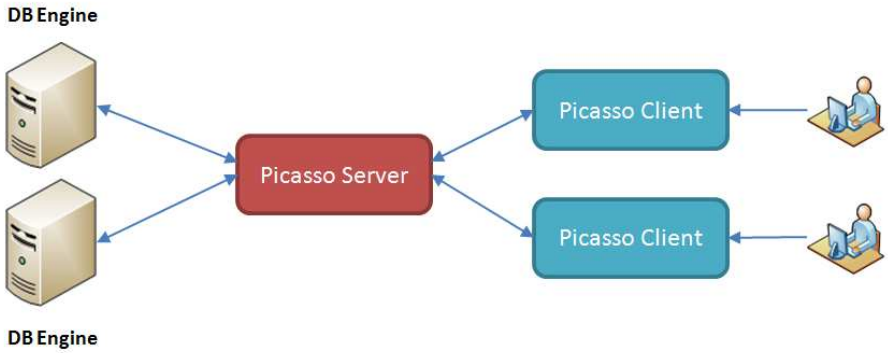


Figure 6.1: Picasso Architecture

the user is passed on from the Picasso client to the Picasso server, which handles communication with the database engine and the production of diagrams. The Picasso client is responsible for the visualization of these diagrams. The Picasso server communicates with the database engines through their JAVA interfaces, treating the optimizers as "black boxes". Picasso currently supports DB2, SQL Server, Oracle, Sybase and PostgreSQL.

6.1.1 Algorithm Implementation

The sampling techniques developed for Class I and II optimizers have been implemented in the *Picasso-Server* module of the Picasso tool. Additional modifications for accepting θ_I and θ_L as user input were done in the *Picasso-Client* module.

6.1.1.1 RS_NN

Optimization Step: We generate a random number for each of the different dimensions within the range 0 to $r-1$ for constructing a query point to optimize. The seed used for the random number generator is set to a constant value to ensure the approximations are reproducible.

Inference: To infer a point $q_u(x_1, x_2, \ldots, x_d)$ we start with chessboard distance 1 and gradually increase the distance until we encounter at least one optimized point. Suppose we want to find out the selectivities of each query point at distance l from q_u. We increase each x_i, starting from 1 to l using recursion ensuring that at least

one of the x_i's is set to l, otherwise it will refer to a point at distance $< l$. Reader can refer to the code presented in the Appendix.

6.1.1.2 GS_PQO

As described earlier, in GS_PQO the entire selectivity space needs to be divided into sub-rectangles. To make it generalized for any dimension, a recursive approach was developed. Figure 6.2 illustrates the process for a 3D rectangle. Suppose we want to divide the rectangle whose origin is denoted by the binary string 000. First the points at *Hamming Distance* 1 are looked at i.e. $001, 100, 010, 100$ and mid points of these edges are optimized or inferred. Then the points at hamming distance 2 i.e. $110, 110, 101$ are referred to process the respective mid points. Afterwards the point at *Hamming Distance* 3 i.e. 111 is considered. For a d dimensional rectangle the process goes on till *Hamming Distance* d. To implement this, we first generate all possible permutations of string with 0,1 of length d (dimension), then recursively call the process for each point.

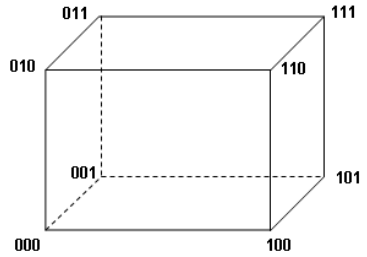


Figure 6.2: Generalization of GS_PQO rectangle split

6.1.1.3 Cost and Cardinality approximation

In both RS_NN and GS_PQO, once all points of the selectivity space are either optimized or inferred, we find out the respective cost and cardinality functions using linear least square regression method. The respective models are fitted for the optimized points. Once the models are derived, we use it to estimate the cost and cardinality values of the inferred points.

Linear Algebra Packages Used. We need to find out solution of matrices to verify whether a point can be expressed as linear combination of the boundary points of a convex polytop and also to implement the linear least square regression technique. We applied LU decomposition for solving matrix of the form $Ax = b$ where A happens to be square matrix and QR decomposition otherwise. Both the LU and QR decomposition method is implemented using the LINPACK [26] routines DGEFA, DGEDI, DQRDC and DQRSL. DGEFA computes the LU decomposition by Gaussian elimination and DGEDI computes the determinant and inverse of a matrix using factors computed by DGEFA. Similarly DQRDC computes the QR decomposition using least squares method, while DQRSL applies the decomposition for finding solution of the form $Ax = b$.

6.1.1.4 Major design changes

Some additional information on algorithm type, optimization overhead etc. needs to be stored for approximate diagram. To serve this purpose we created a new table called ***PicassoApproxMap***. If the diagram type(EXECTYPE) in the table *PicassoQTIDMap* is set to approximate diagram, the additional information is fetched from *PiccasoApproxMap* for the particular QTID. The schema of PiccasoApproxMap is given below.

PicassoApproxMap

- QID : Unique ID (Primary Key) assigned to the approximate diagram.
- SampleSize : Stores percentage sample size required in the approximation process.
- SamplingMode : This saves the specific algorithm id used for approximation e.g. for RS_NN it is 0 and 1 for GS_PQO.
- IdentityErrorBound : Stores the user given threshold on identity error in percentage.
- AreaErrorBound : Stores the user given threshold on location error in percentage.
- FPCMode : Set to 1 if FPC was enabled during approximation.

In the existing code of Picasso 1.0, we added one class called PicassoSampling, which acts as a container for the different sampling and inference techniques, as well as for the methods required for cost and cardinality approximations. The class inheritance diagram and package(*iisc.dsl.picasso.server.sampling*) contents are given below.

1. **PicassoServer**
   (a) **PicassoDiagram**
      i. PicassoSampling (*extends PicassoDiagram*)
         A. RS_NN (*extends PicassoSampling*)
         B. GS_PQO (*extends PicassoSampling*)

(a) **Class Inheritance**

iisc.dsl.picasso.server.sampling

- PicassoSampling.java
- RS_NN.java
- GS_PQO.java
- CostFunction.java
- CardFunction.java

(b) **Sampling Package (Server side)**

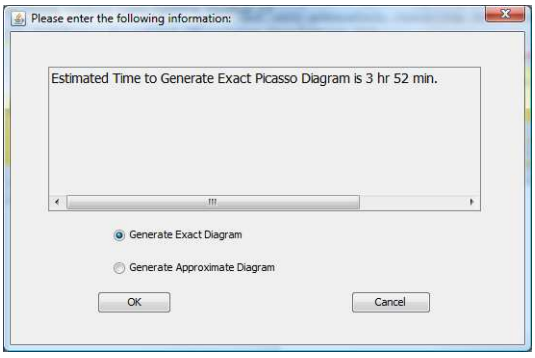
While the above changes were made in the server side code, no such major changes were made at client side, other than the GUI module necessary for retrieving specifications from user regarding approximation, along with the methods to pass the information properly to server side.

6.1.1.5 Customization for Class II optimizers

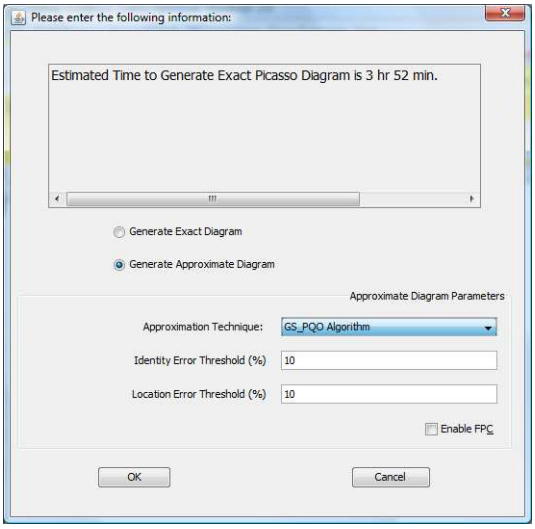
The FPC feature should only be enabled for Class II optimizers, which is ensured through a flag set during initialization of the database connection. FPC is used as mentioned earlier as a tie breaker in the approximation process and during generation of cost and cardinality diagram. We save the abstract plan information e.g. the plan-tree XML for SQL Server corresponding to each plan so that the same skeleton can be used to perform FPC later. The implementation steps for the approximation works in a manner shown in Table 6.1.

Class I	Class II
1. Run algorithms for Optimization + Inference (terminate with ϵ_I and ϵ_L estimators). 2. Find Cost and Card function. 3. Set Cost and Card for each inferred point.	1. Run algorithms for Optimization + Inference (terminate with ϵ_I estimator, break ties with FPC and verify with ϵ_L estimators). 2. Use FPC to calculate Cost and Card for each inferred points.

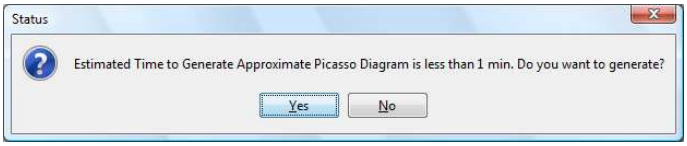
Table 6.1: Steps of approximation algorithms



(c) **Screen 1: Diagram type**



(d) **Screen 2: Algorithm specifications**



(e) **Screen 3: Overhead estimation**

Figure 6.3: User Input Screens (Picasso)

6.1.2 Approximation in Picasso

Following are the steps the user has to follow to generate approximate diagram for Class I and Class II optimizers. Figure 6.3 shows a snapshot of the user query screen.

1. User will be asked regarding the diagram type (Figure 6.3(c)) first.
   - If user chooses to follow the exhaustive approach, the process will start immediately.
   - Else user will be asked specifics concerning the approximation techniques (Figure 6.3(d)).

2. Before the generation process, user will be shown the estimated approximation overhead ((Figure 6.3(e))). User can cancel the process at this point also.

3. After the diagram is generated, user will be informed about the % optimizations done as shown in Figure 6.4(a).

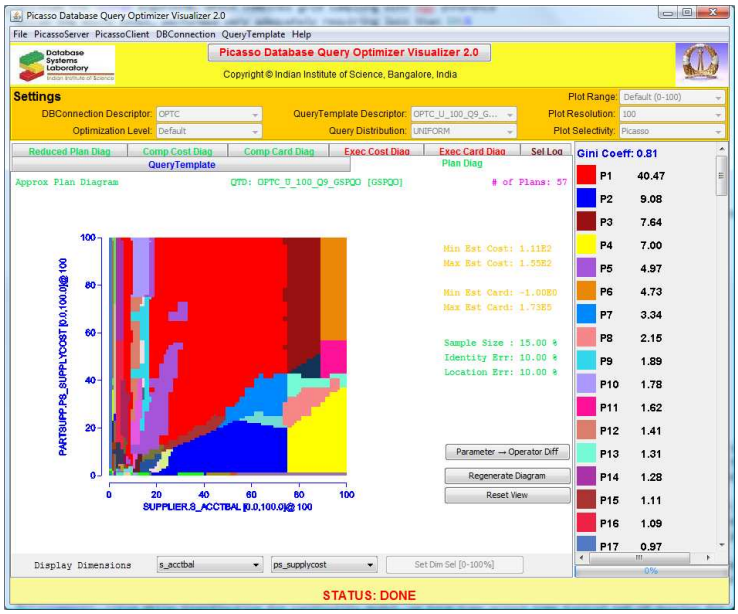
4. Later user can access any approximate diagram from the template list, where each approximate diagram will have "(A)" beside their name as shown in Figure 6.4(b).

Our approximate diagrams do support reduction and other post processing operations which are supported by an exact diagram generated through the default brute-force technique.

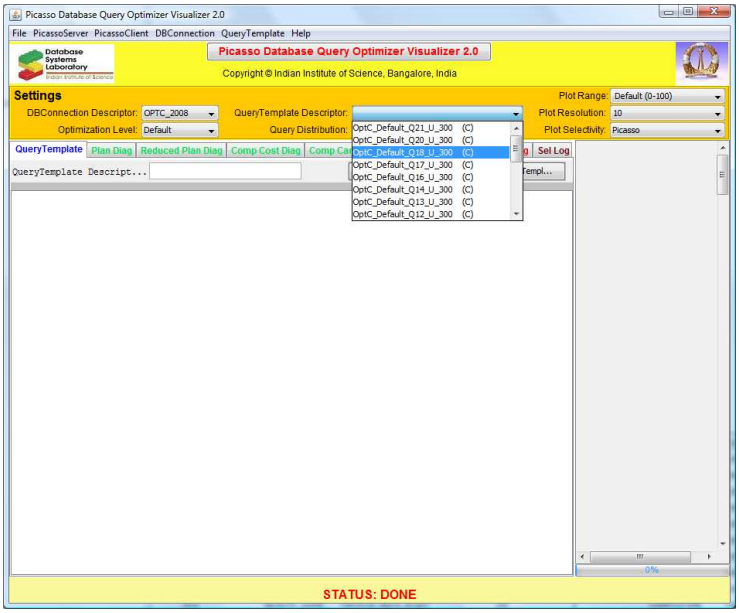
6.2 PostgreSQL

The approximation algorithms for Class III optimizers were implemented and tested in PostgreSQL v8.3.6. In this section we briefly discuss the code changes made in PostgreSQL to implement FPC, PRL, *PlanFill* and *Relaxed-PlanFill*. First we give an overview of the important data structures of PostgreSQL and discuss how we have used them.

Our implementation mainly was concentrated on planner/optimizer part of PostgreSQL.



(a) **The user screen shot after the approximation diagram is generated**



(b) **The dropdown entries for approximate diagrams**

Figure 6.4: Approximation Interface (Picasso)

Important data structures

Path: An access path for a base or join relation.

RelOptInfo: Per-relation information for planning/optimization. Each *RelOptInfo* has

a pointer to the cheapest access path.

RestrictInfo: For each AND sub-clause of a restriction condition (WHERE or JOIN/ON clause) these datastructures are created. Since the restriction clauses are logically ANDed, one can use any one of them or any subset of them to filter out tuples, without having to evaluate the rest. The *RestrictInfo* node itself stores data used by the optimizer while choosing the best query plan. Each *RelOptInfo* may have one or more *RestrictInfo* data structure denoting the restrictions imposed on it.

Plan: This is final plan-tree structure.

These are the basic data structures used by PostgreSQL while planning/optimization defined in the header file *relation.h*.

6.2.1 Foreign Plan Costing

To implement the FPC feature in PostgreSQL we had to divide the code into three different parts as mentioned below,

1. Replacing remote points selectivity in the *Path*.
2. Recosting the modified *Path* till Join-Root.
3. Recosting the stem part of the *Path* i.e. from Join-Root till Plan-Root.

Major Design Considerations Our design choices were based on the existing PostgreSQL code structure.

1. For our purpose, re-costing a plan-tree would have been easier if we could call the FPC functions for the *Plan* structure. However, many of the cost related parameters required for efficient re-costing e.g. RestrictInfo etc present in *Path* are not available in *Plan* structure. Therefore, we call the main FPC function for *Path* structure.
2. While implementing the code we encountered certain issues specific to PostgreSQL's costing functions. We discovered that for the first time PostgreSQL encounters a

certain restriction clause e.g. $A.age = B.age$, it caches some values and reuses it for all the later occasions. This cached values are very specific to the query parameters and in our case we cannot reuse the same cached values while recosting the plan at some remote point. The final plan tree cost was found to be very sensitive to slightest change of these cached values. That lead us to save some extra paths (it may not be part of the query plan tree) which had to be explicitly recosted along with the main plan tree. We saved extra paths for each Join Node (RelOptInfo), Hash-Join Node (RestrictInfo), Bitmap-Heap Scan Path (BitmapHeapPath) and Index Path (IndexPath). Changes are made in *relation.h*, *pathnode.c*, *indxpath.c* and *joinpath.c* to take care of the extra baggage, which is used later in *fp_costsize.c*.

These extra operations actually increases the FPC overhead roughly by 25%, still a single FPC takes roughly $\frac{1}{100}$ of optimization time, which seems sufficiently fast for practical purposes.

3. We only considered costing dependent paths since those paths will get affected with changing selectivities.

The main costing functions are defined in *fp_costsize.c* which hosts similar functions as *costsize.c* tailored for FPC. Current implementation of FPC is not available as an API.

6.2.2 Second Best Plan

We implemented the *Additive Second Best Plan Search* as mentioned in Section 3.4.1. While implementing these feature we had to stop pruning of paths completely. Infact in various cases we had to explicitly enumerate paths getting pruned or not considered at all otherwise by PostgreSQL.

1. To find out 2nd best access-path at each intermediate node we had to stop all kind of pruning of paths. After all paths are generated we scan the list of paths twice and extract the two best paths. Note that by avoiding sorting we improve the efficiency of the code.

2. Two kinds of paths are being propagated upward in the default DP-Tree. First one is the cheapest cost path and second one is the *interesting paths* (which can be more than one in number according to their interesting order). Now if any of the interesting paths becomes cheapest at root, the 2nd cheapest path may get pruned along its path in the DP-tree. Therefore, with each interesting path we have to propagate its 2nd best sibling too. Hence we are actually passing more than two paths ($2 + 2 \times no\ of\ interesting\ paths$) per node to the next higher level in the DP-lattice.

3. The inheritance of cost from best path to other can only be done upon paths having same interesting orders. Note that, we can also inherit the stem-cost in the same manner.

These changes were made in *allpaths.c*, *joinpath.c*, *indxpath.c* and *planner.c*.

6.2.3 *PlanFill*

The main *PlanFill* code is written in *planner.c*. We start by optimizing q and then in the first-quadrant we search for selectivities till which the current plan's cost (found through FPC) is less than or equal to its second best plan using Flood-fill algorithm. The algorithm is given in Figure 6.5.

6.2.4 FPC and PRL as API

We modified the code related to parser (*gram.y* and *keywords.c*) in PostgreSQL to request for PRL from Picasso. We also had to modify the structures related to "EXPLAIN" statement for the same. Now instead of only EXPLAIN user needs to write "EXPLAIN PRL2" to get the second best plan. The second best plan is appended to the best one after a line demarcation. Similarly for FPC request we save the original plan and path and recost them on demand. We are capable of handling 3 selectivity predicates. The modified explain request for FPC is "EXPLAIN FPC ($< Relation2\ ID >, < Foreign\ Selectivity1 >, < Relation2\ ID >, <$

Flood-fill-By-Plan(QueryPoint q,Plan p_1,Cost c_1,Cost c_2)

1. Set Q to the empty queue.
2. Add q to Q.
3. For each element $q_{(x,y)}$ of Q, /*(x, y) denotes the coordinates of the query point in 2D*/
4. For $j = 1$ to $r - x$, /*PlanFill in row order first*/
5. If $q_{(x+j,y)}$ is already assigned with a plan: continue;
6. Perform FPC of $p_{(x,y)}$ at $q_{(x+j,y)}$, suppose the cost is $c_{(x+j,y)}$.
7. If $c_{(x+j,y)} < c_2$: Set $p_{(x+j,y)} = p_{(x,y)}$
8. else break;
9. For each node from (x, y) to $(x + j, y)$ i.e. for $i = 0 : j$, /*searching for valid points in column order */
10. Perform FPC of $p_{(x+i,y)}$ at $q_{(x+i,y+1)}$, suppose the cost is $c_{(x+i,y+1)}$.
11. If $c_{(x+i,y+1)} < c_2$: Set $p_{(x+i,y+1)} = p_{(x,y)}$ and add $q_{(x+i,y+1)}$ in Q.
12. Continue looping until Q is exhausted.
13. Return;

Figure 6.5: The Flood-fill algorithm used in *PlanFill*

$Foreign\ Selectivity2 >, < Relation3\ ID >, < Selectivit3 >$". The "Relation ID" is the unique id assigned by PostgreSQL to the base relations. User can easily find that out by keeping the "DEBUG-PRINT" feature enabled.

Chapter 7

Conclusions

We have investigated in this thesis the efficient generation of approximate *Optimizer Diagrams*, a key resource in the analysis and redesign of modern database query optimizers. Based on the optimizer's API capabilities, we made a partitioning into three different classes of optimizers, and developed appropriate approximation techniques for each class. For Class I, which only provides the optimal plan, our experimental results showed that the GS_PQO algorithm, which combines grid sampling with PQO inference at the micro level, performed very adequately requiring less than 15% overheads as compared to the exhaustive approach, for an error bound of 10%. These overheads came down to 10% when the same algorithm was used in Class II optimizers, due to their additional FPC feature. Finally, for Class III systems, we proved that the *PlanFill* algorithm produced zero errors and was generally able to do so incurring overheads of less than 10%. However, it performs poorly for query templates that have the second-best plan being very close to the optimal choice over an extended region. Finally, the *Relaxed-PlanFill* algorithm traded error for performance, and was able to satisfy the 10% error bound with less than 5% optimizations. It was also able to adequately handle the problem query templates of *PlanFill*. We have also presented methods of approximating cost and cardinality diagrams for Class I optimizers, and showed that approximation error is sufficiently low for a wide range of query templates. Class II and Class III can produce the exact cost and cardinality diagrams with regard to the approximate plan diagram, utilizing the FPC

feature along with the plan diagram generation.

In summary, our work has shown that it is indeed possible to efficiently generate close approximations to high-dimension and high-resolution *Optimizer diagrams*, with typical overheads being an *order of magnitude lower* than the brute-force approach. We hope that our results will encourage all database vendors to incorporate the foreign-plan-costing and/or plan-rank-list features in their optimizer APIs.

7.1 Future Work

The work that has been presented in the thesis can be extended in following ways:

1. Our algorithms feature tuning parameters that have been set after considerable empirical testing. These settings may be a function of the specific optimizer engines and database environments assessed in our experiments – in our future work, we plan to investigate the portability of these settings over a broader spectrum of engines and environments.

2. We also intend to explore alternative inside-engine speedup technique that does not involve approximation. Following are two such ideas:

   (a) We can produce the picture starting from the top-right corner and moving towards the origin. At each query optimization, we can use the lowest cost in the first quadrant with respect to this query as a "pilot" pruning mechanism in the DP routine. Such an idea had been proposed in the context of standard single-query optimization about two decades ago, but it had proved unworkable since the random pilot plan chosen was usually very weak in terms of its pruning ability. However, in the plan diagram context, this shortcoming may not hold to the same extent since we are using a pilot that is expected to be close to the optimal.

   (b) We can also speedup plan diagram generation through re-use of plan computations. The idea is to cache the cost and cardinality results of a limited

set of sub-plans during a subset of the query optimizations, and reuse this information during the DP process for subsequent query optimizations.

3. Modern optimizers do not support FPC and PRL. We would like to incorporate these features as an optimizer API into the publicly available optimizer of **PostgreSQL**. Furthermore, using these API, we wish to incorporate Class III approximation algorithms in Picasso.

4. We would also like to investigate performance of these approximation algorithms in much higher dimension e.g. 5D or above.

Bibliography

[1] S. Acharya, P.B. Gibbons and V. Poosala, "Congressional Samples for Approximate Answering of Group-By Queries", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1999.

[2] S. Acharya, P.B. Gibbons, V. Poosala and S. Ramaswamy, "The Aqua Approximate Query Answering System", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1999.

[3] G. Antonshenkov, "Dynamic Query Optimization in Rdb/VMS", *Proc. of 9th IEEE Intl. Conf. on Data Engineering (ICDE)*, April 1993.

[4] B. Babcock, S. Chaudhuri and G. Das, "Dynamic Sample Selection for Approximate Query Processing", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2003.

[5] A. Betawadkar, "Query Optimization with One Parameter", *Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur*, February 1999.

[6] L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone, "Classification and regression trees", Wadsworth Inc, 1984.

[7] M. Charikar, S. Chaudhuri, R. Motwani and V. Narasayya, "Towards Estimation Error Guarantees for Distinct Values", *Proc. of 19th ACM Symp. on Principles of Database Systems (PODS)*, 2000.

[8] S. Chaudhuri, G. Das and V. Narasayya, "A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2001.

[9] S. Chaudhuri, G. Das and V. Narasayya, "Optimized Stratified Sampling for Approximate Query Processing", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2007.

[10] S. Chaudhuri, G. Das, M. Datar, R. Motwani and V. Narasayya, "Overcoming Limitations of Sampling for Aggregation Queries", *Proc. of 9th IEEE Intl. Conf. on Data Engineering (ICDE)*, 2001.

[11] S. Chaudhuri, R. Motwani and V. Narasayya, "Random Sampling for Histogram Construction: How much is enough?", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[12] S. Chaudhuri, G. Das and U. Srivastava, "Effective Use of Block-Level Sampling in Statistics Estimation" *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 2004.

[13] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems", *Proc. of 17th ACM Principles of Database Systems (PODS)*, June 1998.

[14] F. Chu, J. Halpern and J. Gehrke, "Least Expected Cost Query Optimization: What Can We Expect", *Proc. of 21st ACM Symp. on Principles of Database Systems (PODS)*, May 2002.

[15] F. Chu, J. Halpern and P. Seshadri, "Least Expected Cost Query Optimization: An Exercise in Utility", *Proc. of 18th ACM Symp. on Principles of Database Systems (PODS)*, May 1999.

[16] S. Cohen, G. Dror, E. Ruppin, "Feature Selection via Coalitional Game Theory", *Proc. of Neural Computation*, July 2007.

[17] R. Cole and G. Graefe, "Optimization of Dynamic Query Evaluation Plans", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1994.

[18] Harish D., P. Darera and J. Haritsa, "On the Production of Anorexic Plan Diagrams", *Proc. of 33rd Intl. Conf. on Very Large Data Bases (VLDB)*, September 2007.

[19] Harish D., P. Darera and J. Haritsa, "Identifying Robust Plans through Plan Diagram Reduction", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2008.

[20] Pooja N. Darera, "Reduction of Query Optimizer Plan Diagrams", *Master's Thesis, Dept. of SERC, Indian Institute of Science, http://dsl.serc.iisc.ernet.in/publications/thesis/pooja.pdf*, August 2007.

[21] Gautam Das, "Sampling Methods in Approximate Query Answering Systems", *Invited Book Chapter, Encyclopedia of Data Warehousing and Mining*, 2005.

[22] G. Das and J. Haritsa, "Robust Heuristics for Scalable Optimization of Complex SQL Queries", *Proc. of 23rd IEEE Intl. Conf. on Data Engineering (ICDE)*, April 2007.

[23] A. Deshpande, Z. Ives and V. Raman, "Adaptive Query Processing", *Foundations and Trends in Databases*, 2007.

[24] A. Dey, S. Bhaumik, Harish D. and J. Haritsa, "Efficiently Approximating Query Optimizer Plan Diagrams", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB)*, August, 2008.

[25] A. Dey, S. Bhaumik, Harish D. and J. Haritsa, "Efficient Generation of Approximate Plan Diagrams", *Tech. Rep. TR-2008-01, DSL/SERC, Indian Inst. of Science, http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2008-01.pdf*, 2008.

[26] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, "LINPACK Users' Guide", *SIAM, Philadelphia*, 1979.

[27] R. O. Duda, P. E. Hart and D. G. Stork, "Pattern Classification (2nd Edition)", *John Wiley & Sons*, 2006.

[28] M. Ester, H.P. Kriegel, J. Sander and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", *Proc. of 2nd Intl Conf on Knowledge Discovery and Data Mining (KDD)*, 1996.

[29] S. Ganguly, P.B. Gibbons, Y Matias and A. Silberschatz, "Bifocal Sampling for Skew-Resistant Join Size Estimation", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[30] S. Ganguly and R. Krishnamurthy, "Parametric Query Optimization for Distributed Databases based on Load Conditions", *Proc. of COMAD Intl. Conf. on Management of Data*, December 1994.

[31] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms", *Proc. of 24th Intl. Conf. on Very Large Data Bases (VLDB)*, August 1998.

[32] V. Ganti, M. Lee and R. Ramakrishnan, "ICICLES: Self-Tuning Samples for Approximate Query Answering", *Proc of 26th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2000.

[33] A. Ghosh, J. Parikh, V. Sengar and J. Haritsa, "Plan Selection based on Query Clustering", *Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2002.

[34] R. Gonzalez and R. Woods, "Digital Image Processing", *Pearson Prentice Hall*, 2007.

[35] L. A. Goodman, "On the estimation of the number of classes in a population", *Annals of Math. Stat. 20*, 1949.

[36] G. Graefe and D. DeWitt, "The Exodus Optimizer Generator", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1987.

[37] G. Graefe and W. McKenna, "The Volcano optimizer generator: Extensibility and efficient search", *Proc. of 9th IEEE Intl. Conf. on Data Engineering (ICDE)*, April 1993.

[38] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Survey, 25.2, pp.73-170*, June 1993.

[39] P. Haas, J. Naughton, S. Seshadri and L. Stokes, "Sampling-Based Estimation of the Number of Distinct Values of an Attribute", *Proc. of 21st Intl. Conf. on Very Large Databases (VLDB)*, 1995.

[40] P. Haas, J. Naughton and A. Swami, "On the Relative Cost of Sampling for Join Selectivity Estimation", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[41] P. Haas and L. Stokes, "Estimating the number of classes in a finite population", *In Journal of the American Statistical Association, 93*, 1998.

[42] P. Haas and A. Swami, "Sequential Sampling Procedures for Query Size Estimation", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1992.

[43] P. J. Haas, J. F. Naughton, S. Seshadri and A. N. Swami, "Fixed-precision estimation of join selectivity", *Proc. of 12th ACM Symp. on Principles of Database Systems(PODS)*, May 1993.

[44] Y. C. Ho, "Perturbation analysis: Concepts and algorithms", *Proc. of Winter Simulation Conference*, 1992.

[45] W.C. Hou, G. Ozsoyoglu, and B. K. Taneja, "Statistical estimators for relational algebra expressions", *Proc. 7th ACM Symp. on Principles of Database Systems(PODS)*, March 1988.

[46] W. C. Hou, G. Ozsoyoglu, and B. K. Taneja, "Processing aggregate relational queries with hard time constraints", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1989.

[47] A. Hulgeri and S. Sudarshan, "Parametric Query Optimization for Linear and Piecewise Linear Cost Functions", *Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2002.

[48] A. Hulgeri and S. Sudarshan, "AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions", *Proc. of 29th Intl. Conf. on Very Large Data Bases (VLDB)*, September 2003.

[49] Y. E. Ioannidis and Y. C. Kang, "Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1991.

[50] Y. Ioannidis, R. Ng, K. Shim and T. Sellis, "Parametric Query Optimization", *Proc. of 18th Intl. Conf. on Very Large Data Bases (VLDB)*, August 1992.

[51] M. Jarke and J. Koch, "Query optimization in database systems" *ACM Computing Surveys, 16.2, pp.111-152*, June 1984.

[52] N. Kabra and D. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1998.

[53] A. Keinan,B. Sandbank,C. C. Hilgetag, I. Meilijson, E. Ruppin, "Axiomatic Scalable Neurocontroller Analysis via the Shapley Value", *Proc. of Artificial Life*, July 2006.

[54] D. Kossmann and K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms", *Proc of. ACM Trans. on Database Systems (TODS)*, December 2000.

[55] R. Lipton and J. Naughton, "Query size estimation by adaptive sampling", *Proc. of 9th ACM Symp. on Principles of Database Systems(PODS)*, 1990.

[56] R. Lipton, J. Naughton, and D. Schneider, "Practical selectivity estimation through adaptive sampling", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1990.

[57] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh and M. Cilimdzic, "Robust Query Processing through Progressive Optimization", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 2004.

[58] J. Melton and A. Simon, "Understanding The New SQL: A Complete Guide", *Morgan Kaufmann*, May 1993.

[59] R. B. Myerson, "Game Theory: Analysis of Conflict", *Harvard University Press*, 1991.

[60] F. Olken, "Simple Random Sampling from Relational Databases", *Proc. of 12th Intl. Conf. on Very Large Data Bases (VLDB)*, August 1986.

[61] V. Poosala, Y. Ioannidis, P. Haas and E. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[62] V. Prasad, "Parametric Query Optimization: A Geometric Approach", *Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur*, April 1999.

[63] S. Rao, "Parametric Query Optimization: A Non-Geometric Approach", *Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur*, March 1999.

[64] N. Reddy and J. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers", *Proc. of 31st Intl. Conf. on Very Large Data Bases (VLDB)*, August 2005.

[65] N. Reddy, "Next-Generation Relational Query Optimizers", *Master's Thesis, Dept. of CSA, Indian Institute of Science, http://dsl.serc.iisc.ernet.in/publications/thesis/naveen.pdf*, June 2005.

[66] F. Reiss and T. Kanungo, "A Characterization of the Sensitivity of Query Optimization to Storage Access Cost Parameters", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[67] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, "Efficient and Extensible Algorithms for Multi Query Optimization", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[68] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price, "Access Path Selection in a Relational Database System", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1979.

[69] E. Shekita and H. Young, "Iterative Dynamic Programming", *IBM Tech. Report*, 1998.

[70] A. Shlosser, "On estimation of the size of the dictionary of a long text on the basis of a sample", *Engrg. Cybernetics 19*, 1981.

[71] A. Silberschatz, H. Korth and S. Sudarshan, "Database System Concepts", *McGrawHill*, 1997.

[72] P. Tan, M. Steinbach and V. Kumar, "Introduction to Data Mining", *Addison-Wesley*, 2005.

[73] F. Waas and C. Galindo-Legaria, "Counting, enumerating, and sampling of execution plans in a cost-based query optimizer", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[74] *http://dsl.serc.iisc.ernet.in*

[75] *http://en.wikipedia.org/wiki/Flood_fill*

[76] Picasso Database Query Optimizer Visualizer, *http://dsl.serc.iisc.ernet.in/projects/PICASSO/picasso.html*

[77] *http://en.wikipedia.org/wiki/Bellman_equation*

[78] *http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc34982_1500/html/mig_gde/BABIFCAF.htm*

[79] *http://postgresql.org*

[80] *http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/t0024533.htm*

[81] *http://msdn2.microsoft.com/en-us/library/ms189298.aspx*

[82] *http://www.tpc.org/tpcds*

[83] *http://www.tpc.org/tpch*

Appendix A

Coding Details

In this section we have listed the n-dimensional version of the algorithms described and also the implementation procedure of different techniques mentioned earlier in the report.

A.1 RS_NN

We describe the implementation procedure of the NN-Interpolation and Low Pass Filtering technique mentioned so far.

NN-Interpolation As we have mentioned earlier the interpolation technique relies on the nearest-neighbor approach to look for a suitable plan around as a candidate plan for a non-sampled point. The algorithm listed in the Figure A.1 is invoked for each non sampled point. The implementation procedure is described in Fig A.1.

From here onwards we have used d to represent the dimension of the diagram. The variable *dist* is used to set the chessboard distance at which we are interested in finding the neighbors e.g. if $dist = 4$ then the function *recursiveNN()* derives all possible offsets required to find out neighbors at a particular chessboard distance. The variable $dimPresent$ is used to avoid generating offsets for neighbors at a lesser distance i.e. if $dist = 4$ then we shouldn't generate for $dist = 3, 2 or 1$. This is achieved if at least one of the coordinates of a offset is equal to *dist*. So $dimPresent$ is used to implement that

by forcefully turning the lowest dimension to *dist* if none of the higher dimension is set to so. Then we add these offsets with the coordinates of non-sampled point X and apply the interpolation technique thereafter.

Low-Pass We run one iteration of Low-Pass Filter on the approximate diagram to remove jagged edges introduced by NN interpolation. We look at all the neighbors at distance 1 from a non-sampled point. This can be done by invoking the *NearestNeighbor* algorithm illustrated in Figure A.1 with $dist = 1$. If any of the neighboring plans occupies more than 50% of points, we assign that plan to the non-sampled point.

A.2 GS_PQO

The n-dimensional GS_PQO algorithm is almost same as described in Section 3.2.2 except the *initial grid sampling* and *rectangle decomposition*. The initial grid sampling employs a simple recursive function *InitialGSPQO* shown in Figure A.2 to optimize the corner points of initial rectangles. In the rectangle decomposition step we need to optimize or interpolate the mid-points of all the $d \cdot 2^{d-1}$ edges of a d dimensional hyper-rectangle and break it into 2^d equal hyper-rectangles. The complete algorithm is illustrated in Figure A.2.

```
//Global variables
   Non-Sampled point: X(x_1, x_2, ..., x_d);
   Dimension: d;
   Distance: dist;
   Array of length d: DimVar;
   boolean dimPresent;
   Queue Q;
NearestNeighbor()
    1. dist = 1;
    2. Call recursiveNN(d); //Recursively check neighbors at distance dist
    3. if Q is not NULL
    4.       Assign plan P occupying maximum points to X.
    5.       Return;
    6. dist++;
    7. Go to Step 2;
recursiveNN(Depth)
    1. if Depth = 1
    2.       if dimPresent = true,
    3.             for i= -dist to +dist, increment i by 1
    4.                   dimVar[1] = i;
    5.                   doNNJob();
    6.       else
    7.             dimVar[1] = -dist;
    8.             doNNJob();
    9.             dimVar[1] = +dist;
   10.             doNNJob();
   11. else
   12.       for i= -dist to +dist, increment i by 1
   13.             dimVar[Depth] = i;
   14.             dimPresent = false;
   15.             if i = -dist or i = +dist
   16.                   dimPresent = true;
   17.             recursiveNN(Depth-1);
doNNJob()
    1. NN[1...d] : dimvar[1...d] + X[1...d]
    2. if NN[1...d] is a sampled point
    3.       Add NN into Queue Q;
```

Figure A.1: The n-Dimensional RS_NN Interpolation Algorithm

```
//Global Variables
  Integer interval; //Initial GSPQO interval
  Array of length d: DimVar;
  Resolution: res;
GS_PQO (QueryTemplate Q, ErrorBound ε,Dimension d)
 1. ρ_t = ε
 2. InitialGSPQO(d); //Optimize points in the initial low-resolution grid
 3. Calculate the ρ plan density metric for each
    hyper-rectangle with 2^d corners using Equation 3.2.7.
 4. Organize the hyper-rectangles in a max-Heap structure based on their ρ values.
 5. For the hyper-rectangle R_top at the top of the heap
 6.     If ρ(R_top) ≤ ρ_T     stop
 7.     else
 8.         Extract R_top from the heap
 9.         Apply PQO interpolation to the mid-points of qualifying edges of R_top.
            Optimize all the remaining mid-points.
10.         Split R_top into 2^d equal hyper-rectangles.
11.         Compute ρ values for the smaller hyper-rectangles.
12.         Insert the new hyper-rectangles into the heap
13.         Return to 5
14. End Algorithm GS_PQO
InitialGSPQO(Depth d)
 1. if Depth = 1
 2.     For DimVar[1] = 1 to res increment by 1
 3.         Optimize the point DimVar[1 ... d];
 4. else
 5.     For DimVar[d] = 1 to res increment by 1
 6.         InitialGSPQO(d − 1);
```

Figure A.2: The n-Dimensional GS_PQO Algorithm

Appendix B

TPC-H Query Templates

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
from
    part,
    supplier,
    partsupp,
    nation,
    region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_retailprice :varies
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE'
    and ps_supplycost <= (
        select
            min(ps_supplycost)
        from
            partsupp, supplier, nation, region
        where
            p_partkey = ps_partkey
            and s_suppkey = ps_suppkey
            and s_nationkey = n_nationkey
            and n_regionkey = r_regionkey
            and r_name = 'EUROPE'
            and ps_supplycost :varies
    )
order by
    s_acctbal desc,
    n_name, s_name, p_partkey
```

Figure B.1: QT2

```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = 'BUILDING'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_totalprice :varies
    and l_extendedprice :varies
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate
```

Figure B.2: QT3

```
select
    o_orderpriority,
    count(*) as order_count
from
    orders
where
    o_totalprice :varies
    and exists (
        select
            *
        from
            lineitem
        where
            l_orderkey = o_orderkey
            and l_extendedprice :varies
    )
group by
    o_orderpriority
order by
    o_orderpriority
```

Figure B.3: QT4

```
select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= '1994-01-01'
    and o_orderdate < '1995-01-01'
    and c_acctbal :varies
    and s_acctbal :varies
group by
    n_name
order by
    revenue desc
```

Figure B.4: QT5

```
select
    supp_nation,
    cust_nation,
    l_year,
    sum(volume)
from
    (
        select
            n1.n_name as supp_nation,
            n2.n_name as cust_nation,
            YEAR (l_shipdate) as l_year,
            l_extendedprice * (1 - l_discount) as volume
        from
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2
        where
            s_suppkey = l_suppkey
            and o_orderkey = l_orderkey
            and c_custkey = o_custkey
            and s_nationkey = n1.n_nationkey
            and c_nationkey = n2.n_nationkey
            and (
                (n1.n_name = 'FRANCE'
                and n2.n_name = 'GERMANY')
                or (n1.n_name = 'GERMANY'
                and n2.n_name = 'FRANCE')
            )
            and l_shipdate between '1995-01-01'
             and '1996-12-31'
            and o_totalprice :varies
            and c_acctbal :varies
    )as shipping
group by
    supp_nation,
    cust_nation,
    l_year
order by
    supp_nation,
    cust_nation,
    l_year
```

Figure B.5: QT7

```
select
    o_year,
    sum(case
        when nation = 'BRAZIL' then volume
        else 0
    end) / sum(volume)
from
    (
        select
            YEAR(o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
            n2.n_name as nation
        from
            part,
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2,
            region
        where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = 'AMERICA'
            and s_nationkey = n2.n_nationkey
            and p_type = 'ECONOMY ANODIZED STEEL'
            and s_acctbal :varies
            and l_extendedprice :varies
    ) as all_nations
group by
    o_year
order by
    o_year
```

Figure B.6: QT8

```
select
    n_name,
    o_year,
    sum(amount)
from
    (
select
n_name,
YEAR(o_orderdate) as o_year,
l_extendedprice * (1 - l_discount) -
ps_supplycost * l_quantity as amount
from
    part,
    supplier,
    lineitem,
    partsupp,
    orders,
    nation
    where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%green%'
and s_acctbal :varies
and ps_supplycost :varies
) as profit
group by
    n_name,
    o_year
order by
    n_name,
    o_year desc
```

Figure B.7: QT9

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= '1993-10-01'
    and o_orderdate < '1994-01-01'
    and c_nationkey = n_nationkey
    and c_acctbal :varies
    and l_extendedprice :varies
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
order by
    revenue desc
```

Figure B.8: QT10

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct ps_suppkey) as supplier_cnt
from
    partsupp,
    part
where
    p_partkey = ps_partkey
    and p_retailprice :varies
    and ps_suppkey in (
        select
            s_suppkey
        from
            supplier
        where
            s_acctbal :varies
    )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size
```

Figure B.9: QT11

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct ps_suppkey) as supplier_cnt
from
    partsupp,
    part
where
    p_partkey = ps_partkey
    and p_retailprice :varies
    and ps_suppkey in (
        select
            s_suppkey
        from
            supplier
        where
            s_acctbal :varies
    )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size
```

Figure B.10: QT16

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_retailprice :varies
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
            and l_extendedprice :varies
    )
```

Figure B.11: QT17

```
select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    customer,
    orders,
    lineitem
where
    o_orderkey in (
        select
            l_orderkey
        from
            lineitem
        where l_extendedprice :varies
        group by
            l_orderkey having
                sum(l_quantity) > 300
    )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
    and c_acctbal :varies
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate
```

Figure B.12: QT18

```
select
    s_name,
    s_address
from
    supplier,
    nation
where
    s_suppkey in (
        select ps_suppkey
        from partsupp
        where ps_partkey in (
            select p_partkey
            from part
            where p_name like 'forest%'
            )
            and ps_availqty < (
            select 0.5 * sum(l_quantity)
            from lineitem
            where l_partkey = ps_partkey
                and l_suppkey = ps_suppkey
                and l_extendedprice :varies

            )
    )
    and s_nationkey = n_nationkey
    and s_acctbal :varies
    and n_name = 'AMERICA'
order by
    s_name
```

Figure B.13: QT20

```
select
    s_name,
    count(*) as numwait
from
    supplier,
    lineitem l1,
    orders,
    nation
where
    s_suppkey = l1.l_suppkey
    and o_orderkey = l1.l_orderkey
    and o_orderstatus = 'F'
    and exists (
        select
            *
        from
            lineitem l2
        where
            l2.l_orderkey = l1.l_orderkey
            and l2.l_suppkey <> l1.l_suppkey
    )
    and not exists (
        select
            *
        from
            lineitem l3
        where
            l3.l_orderkey = l1.l_orderkey
            and l3.l_suppkey <> l1.l_suppkey
            and l3.l_receiptdate > l3.l_commitdate
    )
    and s_nationkey = n_nationkey
    and s_acctbal :varies
    and l1.l_extendedprice :varies
    and n_name = 'SAUDI ARABIA'
group by
    s_name
order by
    numwait desc,
    s_name
```

Figure B.14: QT21

Appendix C

TPC-DS Query Templates

```
select
    i_item_id,
    avg(ss_quantity) as agg1,
    avg(ss_list_price) as agg2,
    avg(ss_coupon_amt) as agg3,
    avg(ss_sales_price) as agg4
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 = 'M' and cd_marital_status = 'S'
    and cd_education_status = 'College'
    and (p_channel_email = 'N' or p_channel_event = 'N')
    and d_year = 2000 and ss_sales_price <= 100.0
    and i_current_price <= 50.0
group by
    i_item_id
order by
    i_item_id
limit 100;
```

Figure C.1: DSQT7

```
select
    i_item_desc,
    i_category,
    i_class,
    i_current_price,
    sum(ws_ext_sales_price) as itemrevenue,
    sum(ws_ext_sales_price)*100/sum(sum(ws_ext_sales_price))
    over (partition by i_class) as revenueratio
from
    web_sales,
    item,
    date_dim
where
    ws_item_sk = i_item_sk
    and ws_sold_date_sk = d_date_sk
    and d_date between '1998-05-16' and '998-06-16'
    and i_current_price :varies
    and ws_list_price :varies
group by
    i_item_id,
    i_item_desc,
    i_category,
    i_class,
    i_current_price
order by
    i_category,
    i_class,
    i_item_id,
    i_item_desc,
    revenueratio
```

Figure C.2: DSQT12

```
select
    i_item_id,
    i_item_desc,
    s_state,
    count(ss_quantity) as store_sales_quantitycount,
    avg(ss_quantity) as store_sales_quantityave,
    stdev(ss_quantity) as store_sales_quantitystdev,
    stdev(ss_quantity)/avg(ss_quantity) as store_sales_quantitycov,
    count(sr_return_quantity) as_store_returns_quantitycount,
    avg(sr_return_quantity) as_store_returns_quantityave,
    stdev(sr_return_quantity) as_store_returns_quantitystdev,
    stdev(sr_return_quantity)/avg(sr_return_quantity)
as store_returns_quantitycov,
    count(cs_quantity) as catalog_sales_quantitycount,
    avg(cs_quantity) as catalog_sales_quantityave,
    stdev(cs_quantity)/avg(cs_quantity) as catalog_sales_quantitystdev,
    stdev(cs_quantity)/avg(cs_quantity) as catalog_sales_quantitycov
from
    store_sales, store_returns, catalog_sales,
    date_dim d1, date_dim d2, date_dim d3, store, item
where
    d1.d_quarter_name = '2002Q1'
    and d1.d_date_sk = ss_sold_date_sk
    and i_item_sk = ss_item_sk
    and s_store_sk = ss_store_sk
    and ss_customer_sk = sr_customer_sk
    and ss_item_sk = sr_item_sk
    and ss_ticket_number = sr_ticket_number
    and sr_returned_date_sk = d2.d_date_sk
    and d2.d_quarter_name in ('2002Q1','2002Q2','2002Q3')
    and sr_customer_sk = cs_bill_customer_sk
    and sr_item_sk = cs_item_sk
    and cs_sold_date_sk = d3.d_date_sk
    and d3.d_quarter_name in ('2002Q1','2002Q2','2002Q3')
    and ss_list_price :varies
    and cs_list_price :varies
group by
    i_item_id, i_item_desc, s_state
order by
    i_item_id, i_item_desc, s_state
```

Figure C.3: DSQT17

```
select
    i_item_id,
    ca_country,
    ca_state,
    ca_county,
    avg(cs_quantity) agg1,
    avg(cs_list_price) agg2,
    avg(cs_coupon_amt) agg3,
    avg(cs_sales_price) agg4,
    avg(cs_net_profit) agg5,
    avg(c_birth_year) agg6,
avg(cd1.cd_dep_count) agg7
from
    catalog_sales,
    customer_demographics cd1,
    customer_demographics cd2,
    customer,
    customer_address,
    date_dim,
    item
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 cd1.cd_gender = 'F'
    and cd1.cd_education_status = 'Unknown'
    and c_current_cdemo_sk = cd2.cd_demo_sk
    and c_current_addr_sk = ca_address_sk
    and c_birth_month in (3,11,9,5,8,10)
    and d_year = 2000
    and ca_state in ('NC','AK','PA','AK','CA','MA','WV')
    and cs_list_price :varies
    and i_current_price :varies
group by
    i_item_id,
    ca_country,
    ca_state,
    ca_county
order by
    ca_country,
    ca_state,
    ca_county
```

Figure C.4: DSQT18

```
select
    i_brand_id brand_id,
    i_brand brand,
    i_manufact_id,
    i_manufact,
    sum(ss_ext_sales_price) ext_price
from
    date_dim,
    store_sales,
    item,
    customer,
    customer_address,
    store
where
    d_date_sk = ss_sold_date_sk
    and ss_item_sk = i_item_sk
    and d_moy=12
    and d_year=1999
    and ss_customer_sk = c_customer_sk
    and c_current_addr_sk = ca_address_sk
    and substring(ca_zip,1,5) <> substring(s_zip,1,5)
    and ss_store_sk = s_store_sk
    and ss_list_price :varies
    and i_current_price :varies
group by
    i_brand,
    i_brand_id,
    i_manufact_id,
    i_manufact
order by
    ext_price desc,
    i_brand,
    i_brand_id,
    i_manufact_id,
    i_manufact
```

Figure C.5: DSQT19

```
select
    i_item_id,
    i_item_desc,
    s_store_id,
    s_store_name,
    sum(ss_net_profit) as store_sales_profit,
    sum(sr_net_loss) as store_returns_loss,
    sum(cs_net_profit) as catalog_sales_profit
from
    store_sales,
    store_returns,
    catalog_sales,
    date_dim d1,
    date_dim d2,
    date_dim d3,
    store,
    item
where
    d1.d_moy = 4
    and d1.d_year = 1999
    and d1.d_date_sk = ss_sold_date_sk
    and i_item_sk = ss_item_sk
    and s_store_sk = ss_store_sk
    and ss_customer_sk = sr_customer_sk
    and ss_item_sk = sr_item_sk
    and ss_ticket_number = sr_ticket_number
    and sr_returned_date_sk = d2.d_date_sk
    and d2.d_moy between 4 and  4+6
    and d2.d_year = 1999
    and sr_customer_sk = cs_bill_customer_sk
    and sr_item_sk = cs_item_sk
    and cs_sold_date_sk = d3.d_date_sk
    and d3.d_moy between 4 and 4+6
    and d3.d_year = 1999
    and ss_list_price :varies
    and cs_list_price :varies
group by
    i_item_id, i_item_desc, s_store_id, s_store_name
order by
    i_item_id,
    i_item_desc,
    s_store_id,
    s_store_name;
```

Figure C.6: DSQT25

```
select
    i_item_id, avg(cs_quantity) as agg1,
    avg(cs_list_price) as agg2,
    avg(cs_coupon_amt) as agg3,
    avg(cs_sales_price) as agg4
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 = 'M' and cd_marital_status = 'S'
    and cd_education_status = 'College'
    and (p_channel_email = 'N' or p_channel_event = 'N')
    and d_year = 2000 and cs_list_price <= 150.0
    and i_current_price <= 50.0
group by
    i_item_id
order by
    i_item_id;
```

Figure C.7: DSQT26