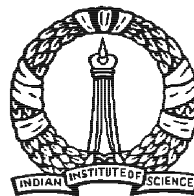
Efficient Generation of Query Optimizer Diagrams

A Project Report
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in
Computer Science and Engineering

By
Sourjya Bhaumik



Computer Science and Automation
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

June 2009

Acknowledgements

I would like to thank my advisor Prof. Jayant Haritsa for helping me to take the first step in the world of scientific research. I would like to sincerely acknowledge his invaluable guidance and encouragement in all forms through out my stay in IISc.

I do convey my gratitude to Atreyee and Harish for their valuable contributions and suggestions in structuring the Joint Paper. I would like to thank all the members of DSL who have made my stay at IISc memorable.

I thank my family and friends for their continued support throughout my career.

Abstract

Given a parameterized n-dimensional SQL query template 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. Similarly, we can define cost diagram and cardinality diagram as the pictorial enumerations of cost and cardinality estimations of the optimizer over the same space. These three diagrams are collectively called "optimizer diagrams". These diagrams have proved to be very useful for the analysis and redesign of modern optimizers but their utility is adversely impacted by the impractically large computational overheads incurred when standard brute-force exhaustive approaches are used for producing fine-grained diagrams on high-dimensional query templates.

In this report, we investigate a variety of intrusive and non-intrusive strategies for efficiently generating computationally expensive optimizer diagrams. The non-intrusive techniques use the query optimizer as a black-box and collectively feature random and grid sampling, as well as classification techniques based on nearest-neighbor and parametric query optimization. The intrusive techniques need changes in the optimizer kernel and leverage the principles of Subplan-Caching, Pilot-Passing and Plan Cost Monotonicity. We evaluate our techniques with a representative set of TPC-H-based query templates on industrial-strength optimizers. The results indicate that our non-intrusive techniques are capable of delivering 90% accurate diagrams while incurring less than 15% of the computational overheads and our intrusive techniques are able to achieve perfect diagrams with around 10% – 70% of the computational overheads when compared to the brute-force exhaustive approach. We have used the Picasso database query optimizer visualizer tool to implement our diagram production strategies and the PostgreSQL query optimizer kernel as the base of our intrusive techniques.

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

3. M. Abhirama, S. Bhaumik, A. Dey, H. Shrimal and J. Haritsa,
   "*Stability-conscious Query Optimization*",
   Technical Report TR-2009-01, DSL/SERC, Indian Institute of Science,
   http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2009-01.pdf

Contents

Abstract **ii**

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

2 Non-Intrusive Algorithms **11**

2.1 Class I Optimizers 11

2.1.1 Random Sampling with NN Inference (RS_NN) 12

2.1.2 Grid Sampling with PQO Inference (GS_PQO) 14

2.2 Class II Optimizers 18

3 Intrusive Algorithms **20**

3.1 The Subplan-Caching based Optimization 21

3.1.1 Subplan-Caching Based Diagram Generation 24

3.2 The Pilot-Based Optimization Technique 25

3.2.1 The Pilot-Passing Technique 26

3.2.2 Pilot-Based Diagram Generation 27

3.3 The PlanFill Algorithm 28

3.3.1 Generating Second-Best Plans 29

3.3.2 Foreign Plan Costing 31

3.3.3 The PlanFill Algorithm 32

4 Implementation Details 35
4.1 Picasso 35
4.2 PostgreSQL 35
5 Experimental Results 38
6 Conclusion and Future Works 42
References 43

List of Figures

1.1 **Example Query Template QT8** . . . 2

1.2 **Example Plan Diagram and Approximate Plan Diagram (QT8)** . 3

1.3 **Algorithm Hierarchy** . . . 7

2.1 **Execution Stages of the RS_NN Algorithm** . . . 14

2.2 **Execution of GS_PQO Algorithm** . . . 15

2.3 ρ **as Plan Richness Indicator** . . . 17

3.1 **Example Query Template and DP Lattice** . . . 21

3.2 **Partially Dependent Nodes** . . . 23

3.3 **Caching and Reuse of Sub-Plans** . . . 25

3.4 **Pilot-Based Diagram Generation** . . . 27

3.5 **Example of DP Lattice** . . . 30

3.6 **The PlanFill Algorithm** . . . 32

List of Tables

1.1 Evaluation Summary 10

5.1 Approximation Efficiency for Class I optimizers with TPC-H ($\theta = 10\%$) [OptCom] 39

5.2 Approximation Efficiency of GS_PQO algorithm for Class II optimizers with TPC-H ($\theta = 10\%$) [OptCom] 39

5.3 Class III (Non-PCM) : Perfect Diagram Efficiency[PostgreSQL] 39

5.4 Class III (PCM) : Perfect Diagram Efficiency[PostgreSQL] 40

5.5 Comparison of Algorithms 40

Chapter 1

Introduction

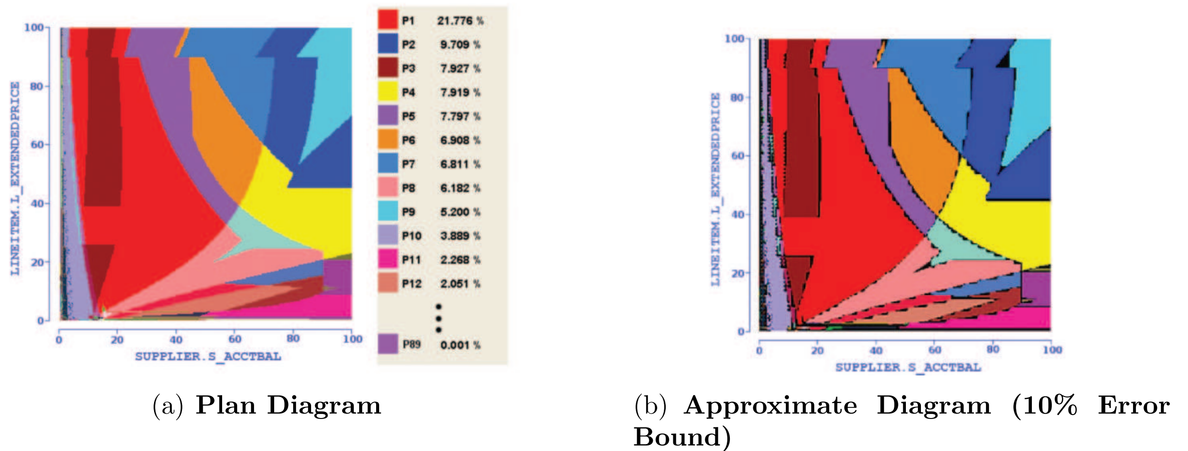
Assuming that the database engine and system configurations are not changing, a query optimizer's execution plan choices are primarily a function of the *selectivities* of the base relations in the query. The concept of a "plan diagram" was introduced in [11] as a color-coded pictorial enumeration of the plan choices of the optimizer for a parameterized query template over the relational selectivity space. For example, consider QT8, the parameterized 2D query template shown in Figure 1, based on Query 8 of the TPC-H benchmark. Here, selectivity variations on the SUPPLIER and LINEITEM relations are specified through the s_acctbal :varies and l_extendedprice :varies predicates, respectively. The associated plan diagram for QT8 is shown in Figure 1.2(a), produced with the Picasso optimizer visualization tool [17] on a popular commercial database engine.

In this picture[1], each colored region represents a specific plan, and a set of 89 different optimal plans, P1 through P89, cover the selectivity space. The value associated with each plan in the legend indicates the percentage area covered by that plan in the diagram – the biggest, P1, for example, covers about 22% of the space, whereas the smallest, P89, is chosen in only 0.001% of the space. A similar representation of costs and cardinalities over the selectivity space is available at [17], for TPC-H benchmark queries on popular commercial optimizers.

[1]The figures in this thesis should ideally be viewed from a color copy, as the gray-scale version may not clearly register the features.

```
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 :varies and l_extendedprice :varies

        ) as all_nations

group by o_year

order by o_year
```

Figure 1.1: Example Query Template QT8



(a) **Plan Diagram**

(b) **Approximate Diagram (10% Error Bound)**

Figure 1.2: Example Plan Diagram and Approximate Plan Diagram (QT8)

Applications of Optimizer Diagrams

Optimizer diagrams have proved to be very useful for the analysis and design of industrial-strength database query optimizers. As evident from Figure 1.2(a), they can be surprisingly complex and dense, with a large number of plans covering the space which is contrary to the conventional assumptions about query optimizers. The reason is that while optimizer behavior on *individual queries* has certainly been analyzed extensively by developers, optimizer diagrams provide a completely different perspective of behav-

ior *over an entire space*, by presenting plan transition boundaries, optimality geometries and cost/cardinality behavior which are in a literal sense, the "big picture" of a query optimizer.

Optimizer diagrams are currently in use at various industrial and academic sites for a diverse set of applications including analysis of existing optimizer behavior; providing visual aid to optimizer regression testing; debugging new query processing features; comparing the behavior between successive optimizer versions; investigating the structural differences between neighboring plans; evaluating the variations in the plan choices made by competing optimizers; etc. Apart from optimizer design support, since plan diagrams 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.

Another particularly compelling utility of optimizer diagrams is that they provide the input to "plan diagram reduction" algorithms. Several significant practical benefits of plan reduction are highlighted in [7]. A detailed study of its application to identify robust plans that are resistant to errors in relational selectivity estimates is available in [8].

Generation of Optimizer Diagrams

The generation and analysis of optimizer diagrams has been facilitated by the Picasso optimizer visualization tool [17]. Given a multi-dimensional SQL query template like QT8 shown in Figure 1 and a choice of database engine, the Picasso tool produces the associated query optimizer diagrams in the following way: For a d-dimensional query template and a plot resolution of r, total r^d queries are generated, with appropriate constants based on their associated selectivities. Then each of these queries are submitted to the query optimizer to be "explained" to obtain their optimal plans. Then a different color is associated with each unique plan and all query points are colored with their associated plan colors. Finally, the rest of the diagram is colored by painting the region around each point with the color corresponding to its plan to produce the complete plan diagram. Note that, as the optimizer also returns the estimated execution cost and result

cardinality along with the optimal plan, the "cost diagram" and "cardinality diagram" [11] are automatically generated while generating the plan diagram.

The above exhaustive approach is acceptable for smaller diagrams which have either low-dimension (1D and 2D) query templates or coarse resolutions (upto 100 points per dimension) or both. However, it becomes impractically expensive for higher dimensions and fine-grained resolutions due to the multiplicative growth in overheads. For example, a 2D plan diagram with a resolution of 1000 on each selectivity dimension, or a 3D plan 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 very useful, their high-dimension and/or fine-resolution versions pose serious computational challenges. Two obvious mechanisms to lower the computational time overheads are – (a) Customize the resolution on each dimension to be domain-specific – for example, coarse resolutions may prove sufficient for categorical data, and (b) 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. Though, perfect diagrams (exactly identical to that obtained with exhaustive approach) are highly desirable, we have observed that it may not be possible to achieve in a generic situation. Specifically, our non-intrusive approaches, which treat the optimizer as a black-box, are unable to deliver perfect diagrams without incurring a overhead almost similar to the exhaustive approach. Therefore, we have to take recourse to generating approximate diagrams instead - the silver lining is that, extensive experimentation with benchmark queries has shown that our non-intrusive techniques can deliver highly accurate optimizer diagrams while incurring overheads almost an order of magnitude lower than the exhaustive approach. Our intrusive techniques on the other hand are able to deliver perfect

optimizer diagrams with significantly lower overheads.

Approximate Plan Diagrams

To evaluate the efficiency of our approximation techniques we define the following metrics. We denote the exact plan diagram as P and the approximation as A. There are two categories of errors that can arise in the plan diagram approximation process:

Plan Identity Error (ϵ_I): This metric considers the possibility of the approximation missing out 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. This error is challenging to control since a majority of the plans appearing in plan diagrams, as seen in Figure 1.2(a), are very small in area and therefore hard to find.

Plan Location Error (ϵ_L): This metric considers the possibility of incorrectly assigning plans to query points in the approximate plan diagram. It is computed as the percentage of incorrectly assigned points in A relative to P. This error is also challenging to control since the plan boundaries, as seen in Figure 1.2(a), can be highly non-linear and are sometimes even irregular in shape.

Approximate Cost and Cardinality Diagrams

The approximation techniques also cause errors in the cost and cardinality diagrams in two ways:

Wrong plan choice: The wrong plan choices induced by the plan diagram approximation process can cause a wrong estimated cost value, if we generate the approximate cost diagram by explicitly costing plans (assigned by the plan diagram approximation process) at each point. Note that, cardinality diagrams are not affected by this type of error as cardinality is not a function of plan.

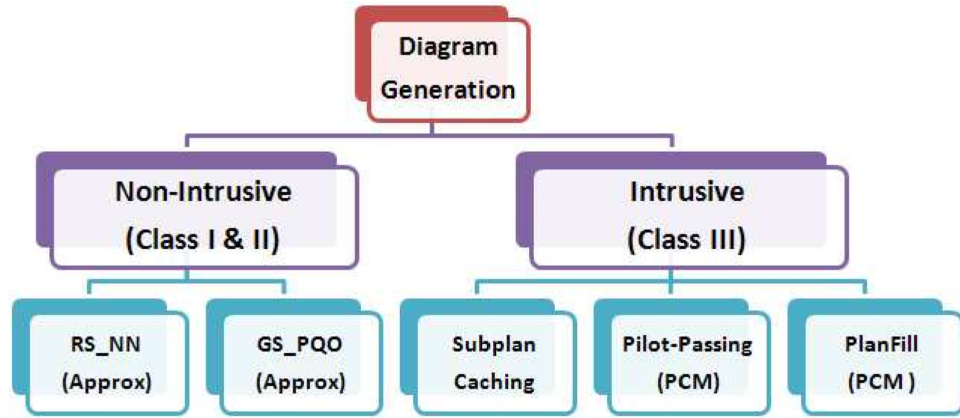


Figure 1.3: Algorithm Hierarchy

Wrong interpolation: Even if the plan diagram is perfect, the associated cost and cardinality diagrams can contain errors, if they are generated through interpolation techniques based on the mathematical models of plan behavior over the parameter space. This interpolation technique is used where we do not have the facility to cost a plan explicitly at any point.

We measure the errors that arise out of the above causes, through a set of metrics including *maximum error* and *root mean square error*. We will not discuss the approximation strategies used for cost and cardinality diagrams, in this thesis. A detailed study of these strategies and their efficiencies is presented in [2].

Diagram Generation Techniques

To show that the ability to reduce overheads directly depends on the plan-related functionalities offered by the optimizer's API, we classify the universe of query optimizers into the following three categories and propose customized diagram generation techniques for each class.

Class I: OP Optimizers. This class refers to the generic cost-based optimizers, where the API only provides the optimal plan (OP), as determined by the optimizer, for a user query. The diagram generation strategies for this class are approximation

algorithms, which are based on a combination of sampling and inference. The sampling techniques include both classical random sampling and grid sampling, while the inference approaches rely on nearest-neighbor (NN) classifiers [14] and parametric query optimization (PQO) [9, 10]. To produce cost (and cardinality) diagrams for this class of optimizers, we rely on linear least square regression method based on mathematical models of plan behavior (and cardinality behavior) over the parameter space.

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 *at any point* in the selectivity space. Specifically, the feature supports the following scenario: "What is the estimated cost of plan p if utilized at query location q?". FPC has become available in the current versions of some industrial-strength optimizers, including DB2 (Optimization Profile), SQL Server (XML Plan), and Sybase (Abstract Plan). The algorithms designed for class I optimizers take advantage of the FPC feature in some situations which are ambiguous to them and thereby provide approximation abilities with lower computational overheads as compared to the class I. The cost and cardinality diagrams for this class can be produced by explicitly costing the plans at the points, where they are assigned by the plan diagram approximation algorithms.

Class III Optimizers: This class of optimizers, provides three additional features apart from the optimal plan and FPC – (a) **"subplan-caching"** strategy, where, the subplans which are predicted to be constant through-out a significant portion of the selectivity space, are cached and reused across multiple optimizations. (b) **"pilot-passing"**, which accepts a cost value with the query, and during optimization process, prunes every path which exceeds that cost value. And (c) **"plan-rank-list" (PRL)**, which along with the optimizer's default choice of best plan, also provides 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 above three strategies collectively represent our *intrusive* set of algorithms and they are

capable of delivering perfect plan diagrams. The cost and cardinality diagrams are automatically generated for class III algorithms during the plan diagram generation process, as will be explained later.

All the algorithms presented in this thesis are shown in 1.3 in a hierarchical form. To the best of our knowledge, the above mentioned three features are not yet available in any of the industrial-strength optimizers. Therefore, we investigate their utilities through our own implementation in PostgreSQL [16] optimizer kernel. Note that, we also implement FPC in PostgreSQL, as this facility is not supported by the optimizer's API.

Evaluation of Diagram Generation Techniques

We have adopted different approaches to evaluate our approximation and perfect diagram generation strategies. We have quantitatively assessed the efficacy of the various approximation strategies, with regard to plan identity and location errors, using a representative suite of multi-dimensional TPC-H-based query templates on leading commercial optimizers. Our experiments consider the scenario where the user expected error is no more than *10 percent* on both plan identities and plan locations. We can achieve this target for Class I (OP) optimizers 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. An example of an approximate diagram (having 10% identity and 10% location error) is shown in Figure 1.2(b), with all the erroneous locations marked in black – as can be seen, the effect of identity error is visually negligible and the location error due to approximation is thinly spread across the diagram and largely confined to the plan transition boundaries. For Class II (OP+FPC) optimizers, a similar error performance is achieved with *only around 10% overhead.* The quality evaluation of cost and cardinality diagrams for class I and class II optimizers, are not presented in this thesis. We request the reader to refer to [2] for a detailed study of the same.

On the other hand, as the diagrams produced by our intrusive techniques contain no error, we evaluate the efficiency of the various strategies through their capability of bringing down computational overheads. The three intrusive techniques mentioned earlier,

are able to achieve the perfect diagrams with around 10% – 70% overheads.

We summarize our evaluation in Table 1.1. As can be seen, our results indicate that – (a) accurate approximations of plan diagrams can indeed be obtained *cheaply and consistently* and (b) perfect diagrams can be obtained with significantly lower overheads than compared to the exhaustive approach. Note that, our algorithms for class I and class II are unable to produce perfect diagrams without optimizing all the query points in selectivity space.

Optimizer Class	**Overheads**	
	Approximation (Bound = 10%)	**Perfect Diagram**
Class I	1% – 15%	100%
Class II	1% – 10%	100%
Class III	–	10% – 70%

Table 1.1: Evaluation Summary

Organization

The remainder of this thesis is organized as follows: The non-intrusive approximation algorithms for class I and class II are presented in Section 2. The intrusive techniques for class III are presented in Section 3. Section 4 outlines the design issues and the implementation challenges. Our experimental framework and performance results are highlighted in Section 5. Finally in Section 6, we outline the future research avenues and conclude this thesis.

Chapter 2

Non-Intrusive Algorithms

As explained earlier, our non-intrusive techniques are basically approximation algorithms. For ease of presentation, we will assume in the following discussion that the query template is 2D – the extension to n-dimensions is straightforward and given in [4]. The exact plan diagram is denoted by P and the approximation as A, with the total number of query points in the diagrams denoted by m. 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). Finally, the plan identity and plan location errors of an approximate diagram are defined as $\epsilon_I = \frac{|P| - |A|}{|P|} * 100$ and $\epsilon_L = \frac{|p_A(q) \neq p_P(q)|}{m} * 100$, respectively. The approximation techniques should ideally ensure that $\epsilon_I \leq \theta_I$ and $\epsilon_L \leq \theta_L$, where θ_I and θ_L are the user-specified bounds on these metrics. In practice, the actual errors should be in the close proximities of user given bounds.

2.1 Class I Optimizers

The approximation procedures for this class of optimizers proceed through interleaved phases of *Optimization* and *Inference*. The random sampling based technique uses statistical estimators, as well as a heuristic to stop the optimization phase whereas the grid

sampling based algorithm relies only on heuristics based on experimental results for the same.

2.1.1 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 phase and then assign plans to all non-optimized points in the inference phase. The optimization phase has two separate passes, each of which is terminated based on our two different error metrics. The stopping criterion for the first pass is based on the identity error metric. To estimate the plan cardinality in the plan diagram we use the classical statistical problem of finding distinct classes in a population [1], from which we obtain the following: Let s samples be taken from the plan diagram, let d_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, d, in the entire plan diagram is in the cardinality range $[d_s, d_{max}]$, where

$$d_{max} = (\frac{m}{s} - 1)f_1 + d_s \tag{2.1}$$

If sampling is iteratively continued until d_s is within θ_I of d_{max}, then it is highly likely that the number of plans found thus far in the sample is within θ_I of d. But our experience, as borne out by the experimental results given in [4], 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 use $\hat{d}_{GEE}$, the guaranteed-error-estimator (GEE) for d, defined in [1] as

$$\hat{d}_{GEE} = (\sqrt{\frac{m}{s}} - 1)f_1 + d_s \tag{2.2}$$

which has an expected ratio error bound of $O(\sqrt{\frac{m}{s}})$.

So the first pass of the RS_NN algorithm optimization phase terminates in two steps as follows: After d_s increases to a value within a $(1 - \delta)$ factor of d_{max}, we continue the

sampling until d_s reaches to within a $(1 - \theta_I)$ factor of $\hat{d}_{GEE}$. The value of δ conducive to good performance results has been empirically determined to be 0.3. The intuition behind this method is that once the gap between d_s and d_{max} has narrowed to a sufficiently small range, then the guaranteed-error-estimator can be used as a reliable indicator of the plan cardinality in the diagram.

The second pass of the optimization phase is terminated based on the location error metric. To estimate the location error, we take recourse to heuristic, which, based on the plans present in the neighborhood of a point, estimates the expected contribution of that point towards the overall location error. The intuition behind this heuristic is presented in [2]. We will not present the details of this heuristic in this thesis.

Inference. After the completion of the sampling phase, the plan choices at the unoptimized points of the plan diagram need to be inferred from the plan choices made at the sampled points. This is done using a Nearest Neighbor (NN) style classification scheme [14]. Specifically, for each unoptimized 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. The distance between two query points is calculated using *Chessboard Distance Metric* which is most suitable, since the transition boundaries between plans often tend to be aligned along the axes of the selectivity space.

Low Pass Filter (LPF). Inference using the NN scheme is well-known to result in boundary errors [14] – in our case, along the plan optimality boundaries. To reduce the impact of this problem, we apply a low-pass filter [5] after the initial inference has assigned plans to all the points in the diagram. The filter operates as follows: For each unoptimized point q_u, all its neighbors (both optimized and unoptimized) 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_u, otherwise the original assignment is retained.

The functioning of the RS_NN algorithm is illustrated in Figure 2.1 – in this set of pictures, each large dot indicates an optimized query point, whereas each small dot

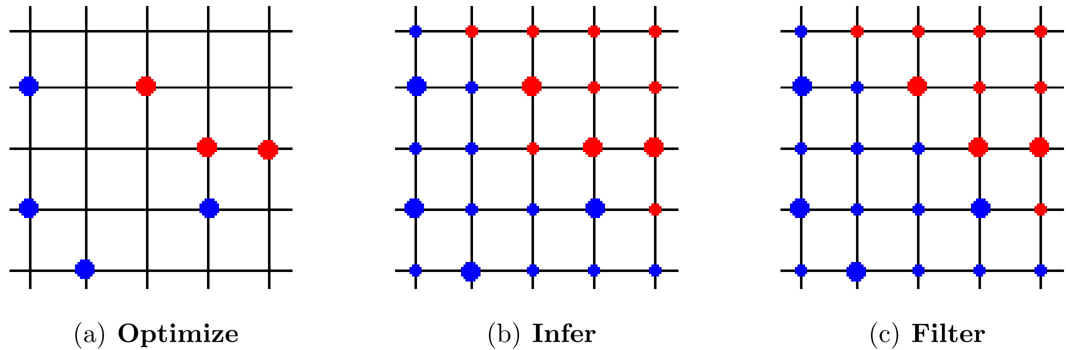


Figure 2.1: Execution Stages of the RS_NN Algorithm

indicates an inferred query point. The initial set of optimized sample query points is shown in Figure 2.1(a), and the NN-based inference for the remaining points is shown in Figure 2.1(b). Applying the LPF filter results in Figure 2.1(c) – note that the center query point, which has an (inferred) red plan in Figure 2.1(b), is re-assigned to the blue plan in Figure 2.1(c).

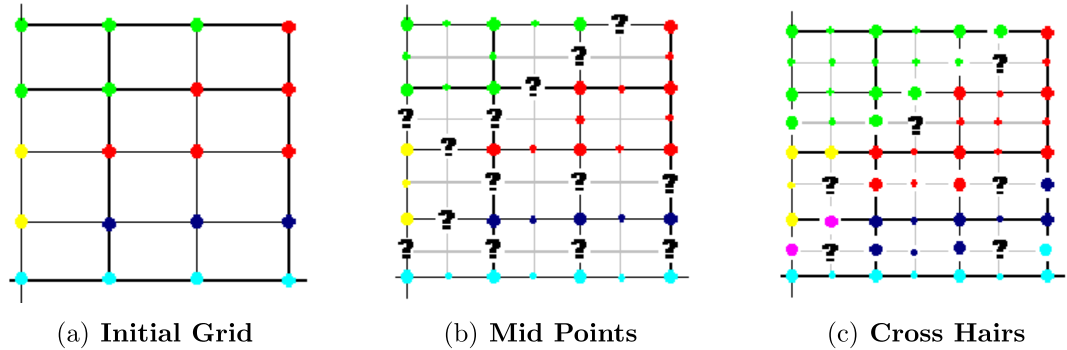
2.1.2 Grid Sampling with PQO Inference (GS_PQO)

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. [9]): "If a pair of points in the selectivity space have the same optimal plan p_i, then all locations along the straight line joining these two points will also have p_i as their optimal plan." Though the observations in [11] indicate that for industrial-strength optimizers, this principle is frequently violated, we are applying PQO only at a "micro-level", that is, within the confines of a small rectangle in the selectivity

space where PQO generally holds.

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 none 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.



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

Figure 2.2: Execution of GS_PQO Algorithm

The progress of the GS_PQO algorithm is illustrated in Figure 2.2 with the same convention. Figure 2.2(a) shows the state after the initial grid sampling is completed. Then, the '?' symbols in Figure 2.2(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 2.2(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 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. 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 compare the *plan operator trees* associated with these plans in order to estimate interior plan density. The detailed study of *Plan Tree Differencing* techniques is given in [3]. We use it as a *black-box* in this thesis.

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 [14] between the trees of the two plans, and is computed as

$$\rho(T_i, T_j) = 1 - \frac{|T_i \cap T_j|}{|T_i \cup T_j|} \tag{2.3}$$

While Equation 2.3 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:

$$\rho(T_1, \ldots, T_n) = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \rho(T_i, T_j)}{\binom{n}{2}} \tag{2.4}$$

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. Our experiments have confirmed that ρ is indeed a good indicator of plan density. As an example, Figure 2.3 shows a plan diagram with a grid and the calculated ρ values for each of the grid rectangles. As can be seen, the ρ values are clearly indicating, which portion of the plan diagram is dense with plans (0.33 value at lower-left rectangle) and which portion is sparse (0.09 value at lower-right rectangle).

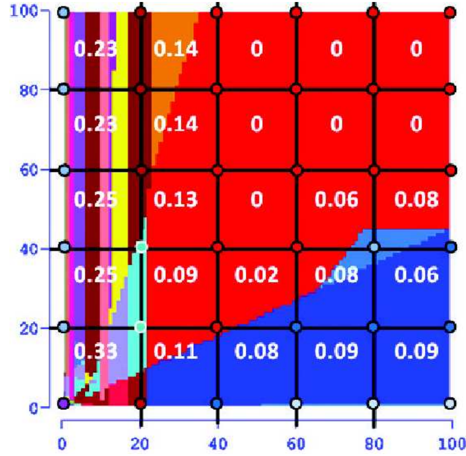


Figure 2.3: ρ as Plan Richness Indicator

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 2.4. 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. The threshold is a function of the θ_I and θ_L bounds given by the user, with lower thresholds corresponding to tighter bounds.

As per the study presented in [4], setting the threshold equal to the identity error bound, i.e. $\rho_t = \theta_I$ (e.g. for $\theta_I = 10\%, \rho_t = 0.1$), is a conservative heuristic that is sufficient to meet user expectations on identity error metric. To tame the location error, we devise another heuristic which estimates the expected location error ($\hat{\epsilon}_L$) based on the inferred points so far. For detailed study of this heuristic, refer to [2]. Therefore, the

initial phase of GS_PQO terminates in two steps. First, on meeting the condition that $\rho(R_{top}) \leq \theta_I$ and then, on ensuring that $\hat{\epsilon}_L \leq \theta_L$.

Final Inference. After both the conditions are met, all the remaining rectangles in the heap are processed in the following way: 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.

2.2 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 unoptimized query point. For example, in the RS_NN approach, if there are multiple nearest neighbors with different plans at the same distance. Similarly, in the final inference phase of GS_PQO approach, where there are unassigned internal points in any rectangle. The strategy followed as far is to make a random choice from the closest neighboring plans. As the success probability in random selection technique decreases with the increasing number of candidates, class I algorithms suffer from large amount of location errors near the plan boundaries and dense regions. The only way to avoid this error is to allow more samples which in turn increases the computational overhead.

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. In cost-based optimizers, this strategy makes perfect sense and significantly helps in reducing the plan-location error, since it enables precise demarcation of the boundaries between plan optimality regions. This means that we can make the thresholds stronger than that of Class I algorithms which will in turn reduce the overhead as the costing of plans is much cheaper than optimizing a query [10].

Chapter 3

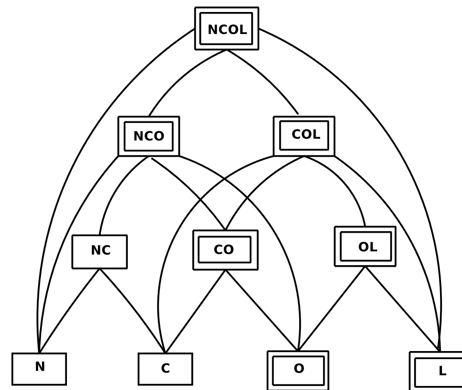
Intrusive Algorithms

We now move on to presenting our intrusive techniques for the Class III optimizers. We present in this section, the customized features we incorporated into the optimizer kernel and the algorithms which leverage them to produce perfect optimizer diagrams.

Specifically, we discuss (a) **Subplan-Caching** technique, which is minimally intrusive in nature and speeds up an individual optimization process to collectively reduce the diagram generation overhead even if all points are optimized, (b) **Pilot-Passing** technique, which requires further more modifications to dynamic programming based optimization process (than the Subplan-Caching technique) and again speeds up the an individual optimization to bring down the overall diagram generation overhead, and (c) **PlanFill** algorithm, which needs significant amount of changes in optimizer and uses the plan-rank-list (PRL) and foreign plan costing (FPC) features to generate perfect diagrams while optimizing only a small subset of points in the selectivity space. Also, the first technique is most generic in the sense that, it does not depend on any assumption about optimizers. The other techniques are however based on a certain assumption about optimizer's cost behavior over the selectivity space.

select C.c_custkey, C.c_name, C.c_acctbal, N.n_name,
C.c_address, C.c_phone
from customer C, orders O, lineitem L, nation N
where C.c_custkey = O.o_custkey
and L.l_orderkey = O.o_orderkey
and C.c_nationkey = N.n_nationkey
and **O.o_totalprice :varies**
and **L.l_extendedprice :varies**

(a) **Query Template**



(b) **DP Lattice**

Figure 3.1: Example Query Template and DP Lattice

3.1 The Subplan-Caching based Optimization

Our first intrusive strategy, the Subplan-Caching technique is minimally intrusive in nature and does not depend on any assumption about the behavior of optimizers. Consider the query shown in Figure 3.1(a), which is the *SPJ* version of the query 10 of TPC-H benchmark. Here, selectivity variations on the *orders* and *lineitem* relations are specified through the O.o_totalprice :varies and L.l_extendedprice :varies predicates, respectively. The DP lattice for this query is shown in Figure 3.1(b), with the nodes (representing different sub-plans) shown as rectangles. The set of relations associated with any node is also shown (e.g. the label *CO* indicates that, the node is considering join between *customer* and *orders*). If *orders* and/or *lineitem* is present in any node, it is shown as double-bordered rectangles. Now, during optimizer diagram generation, we will vary the selectivities of *orders* and *lineitem*, which in turn affect the cardinality estimation (and therefore cost)

of all those double-bordered rectangles. Hence, the choice of optimal sub-plan (or plan at root node) at those nodes may vary across selectivity space. We refer to these type of nodes as the "dependent" nodes. The remaining nodes (e.g. nodes **NC**, **N** and **C** in Figure 3.1(b)) are not affected by the changing selectivity, and we refer to these nodes as the "independent" nodes. The following theorem lays the basis of our Subplan-Caching based diagram generation technique.

Theorem 1 *The choice of optimal sub-plan of an independent node, in DP lattice is constant throughout the selectivity space.*

Proof: Suppose, we are generating plan diagram for a query template. Let X be an independent node in DP lattice. Then by definition, the set of relations associated with X do not have any selectivity predicate (*:varies*). Now, suppose at some point q_1 in the selectivity space, the choice of optimal sub-plan from node X is p_X and the optimal plan at that point is P_{q_1}.

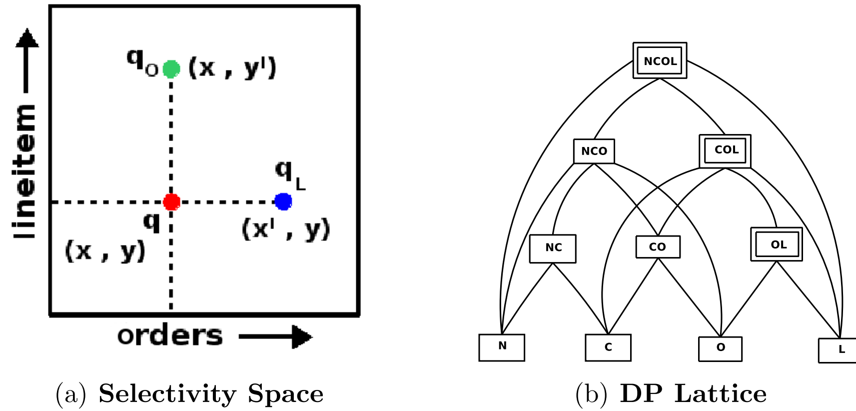
Now, let us assume that, at a different point q_2 in the selectivity space ($q_1 \neq q_2$), the optimal plan is P_{q_2} and the choice of optimal sub-plan from node X is changed to p_X'. As, p_X and p_X' are different sub-plans, their costs will be different. Suppose p_X is cheaper than p_X'. Now, if we replace the sub-plan p_X' with p_X in the plan P_{q_2}, we will obtain another plan for point q_2 which is cheaper than P_{q_2}, due to the additive nature of cost in a plan structure. This new plan is completely valid for q_2 because, the sub-plan p_X does not contain any of the base relations, which determines the position of q_2 in the selectivity space. So, we have constructed a plan for q_2, which is cheaper than the optimal plan, a contradiction. ■

We can leverage this property to generate plan diagrams efficiently, in the following simple way:

Subplan-Caching: During the first optimization, identify all the independent nodes and cache their choice of optimal sub-plans in a global structure, inside optimizer kernel.

Subplan-Reuse: In all successive optimizations, do not compute the sub-plans for any independent node, rather reuse the cached sub-plans.

Unfortunately, the above technique fails to fetch any significant speedup in the optimization process, due to the scarcity of the independent nodes in DP lattice. As we move up in the lattice, almost all of the nodes become dependent, as is clearly illustrated by Figure 3.1(b) (only 3 out 10 nodes are independent). The following observation indicates that, more speedup can be extracted from the Subplan-Caching technique, if we also consider the "partially dependent" nodes.



(a) **Selectivity Space** (b) **DP Lattice**

Figure 3.2: Partially Dependent Nodes

Partially Dependent Nodes. Consider the DP lattice shown in Figure 3.1(b). Among the dependent nodes, there are some members that do not contain both the *orders* and *lineitem* relations. We refer to these nodes as the "partially dependent" nodes. As an example, take the *NCO* node. The selectivity variation of relation *lineitem* will not affect this node, or in other words, the choice of optimal sub-plan of *NCO* node will remain constant, if the selectivity of relation *orders* is not changed. To put the whole thing in perspective, consider the selectivity space shown in Figure 3.2(a) for the query in Figure 3.1(a). When we are optimizing the query q at location (x, y) (x corresponding to *orders* and y to *lineitem*), if we save the sub-plan corresponding to the node *NCO*, we can reuse that for query q_O at location (x, y'). This is because, the selectivity of relation *orders* is same for these two queries. Similarly, if we save the sub-plan for the node L, we can reuse it while optimizing the query q_L at location (x', y). In Figure 3.2(b), only

the fully dependent nodes (for the query in Figure 3.1(a))are shown in double-bordered rectangles. Notice that the total number of independent and partially dependent nodes are now 7 out of 10.

3.1.1 Subplan-Caching Based Diagram Generation

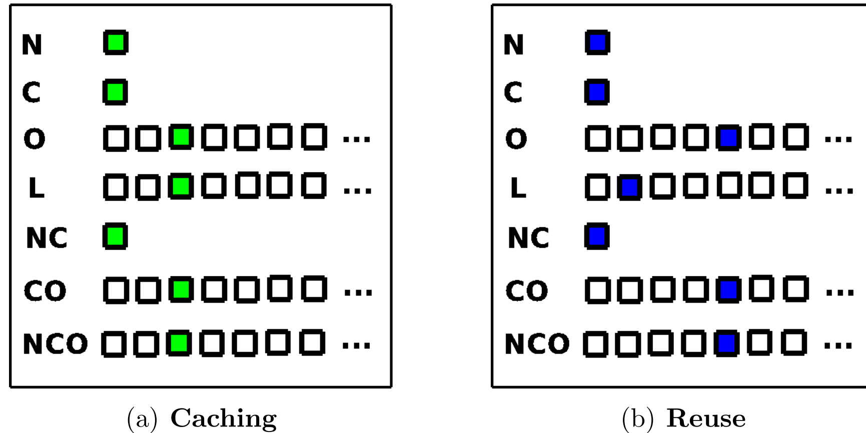
Depending on the above observation, we augment our plan diagram generation process in the following way:

Cache Structure: Keep single placeholders for the independent sub-plans and arrays for partially dependent sub-plans. The size of each array is equal to the resolution (e.g. 100) of the dependent selectivity, for which the plan diagram is being generated.

Initial Caching: First optimize all the diagonal query locations. During the optimization of query at location (x, x), copy independent sub-plans at the corresponding placeholders (if they are not saved already) and copy partially dependent sub-plans at location x of the corresponding arrays.

Diagram Generation: Now optimize rest of the queries in any order and during each optimization reuse the appropriate cached sub-plans. More specifically, during the optimization of query at location (x', y'), get the independent sub-plans from their placeholders and get the partially dependent sub-plans from either x' or y' location of the corresponding arrays.

Consider the example shown in Figure 3.3, which illustrates the state of global cache for the query template of Figure 3.1(a). Figure 3.3(a) highlights the locations (green boxes), where sub-plans are stored when we are optimizing a query at location $(2, 2)$, during initial caching phase. Note that the subplans for NC, N and C will be saved during the optimization of first diagonal point and will never be saved thereafter. Here we show green rectangle for them to clarify our example only. Figure 3.3(b) points out the locations (blue boxes), which are accessed during the optimization of a query at location $(4, 1)$. Note that, the shape of the selectivity space is assumed to be square



(a) **Caching** (b) **Reuse**

Figure 3.3: Caching and Reuse of Sub-Plans

(same resolution on every axis) in the above discussion for the ease of presentation. The extension to the more general rectangular space is straight-forward.

Limitation of Subplan-Caching technique. Though, the above technique is independent from any prior assumption about optimizer's behavior, it has one weakness. Notice that, the performance of Subplan-Caching worsens with the increasing dimensionality of the selectivity space. This is an artifact of decreased number of independent and partially dependent sub-plans in DP, as a large number of dependent nodes at leaf level (relations with selectivity predicate) causes majority of the intermediate nodes to become dependent. Therefore, we suggest to use Subplan-Caching technique for selectivity space with low dimensionality (typically 2D).

3.2 The Pilot-Based Optimization Technique

We now move on to describing the second intrusive algorithm for class III optimizers. This technique needs a few more modifications to the DP based optimization process and relies on the Plan Cost Monotonicity assumption about the optimizer. However, like Subplan-Caching, here also we try to lower the overhead by speeding up an individual optimization process and thereby collectively reduce the diagram generation overhead.

This strategy primarily depends on the Pilot-Passing technique which was first suggested in [12].

3.2.1 The Pilot-Passing Technique

The original idea of speeding up the query optimization process through Pilot-Passing is described below:

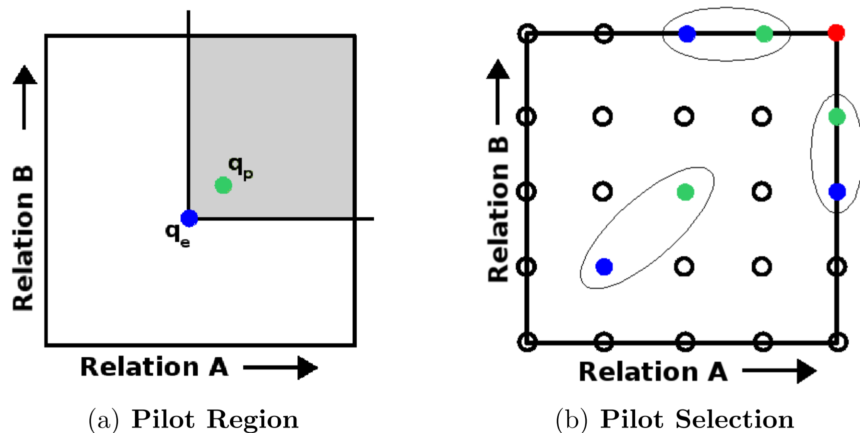
1. Given a query, quickly devise a plan by depth-first-search technique (query optimizer follows breadth-first-search to construct the optimal plan).

2. Take the cost of this plan as the pilot-cost (cost of the optimal plan can never be greater than this).

3. Carry out the normal dynamic programming based breadth-first-search with the exception that, any candidate sub-plan with cost higher than the pilot-cost is pruned and never considered again in the optimization process.

The above technique is predicated on the fact that, as we move up the DP lattice, only additive constants are applied to the sub-plans and therefore cost of a particular sub-plan is always increasing in the optimization process. As the pilot-cost provides an upper-bound on the cost of the final plan at the root of DP lattice, we can safely prune all the candidate sub-plans that have costs higher than the pilot-cost. This should speedup the optimization process, as we will have early pruning of some of the candidates.

Unfortunately, the above technique performs poorly in practice, when implemented on commercial strength optimizers and tested with benchmark queries. The reason is the following: The plan search space is exponentially large and therefore contains many inferior plans, whose cost can be several orders of magnitude away from that of the optimal plan. If our initial choice of plan (obtained through depth-first-search) turns out to be too inferior compared to the optimal plan, we will virtually not get any significant speedup as the costs of almost all candidate sub-plans will be well within the pilot-cost. As a matter of fact, the efficiency of Pilot-Passing technique relies heavily on obtaining an



(a) **Pilot Region** (b) **Pilot Selection**

Figure 3.4: Pilot-Based Diagram Generation

initial plan which is very close to the optimal plan. While, for an isolated optimization, this is very hard to achieve (almost as hard as the actual optimization), we can obtain it very cheaply during optimizer diagram generation. This gives rise to our next perfect diagram generation strategy which uses Pilot-Passing technique as an API.

3.2.2 Pilot-Based Diagram Generation

Consider the relational selectivity space shown in Figure 3.4(a) involving two relations, *Relation A* and *Relation B*, where we are optimizing the point q_e (shown as a blue dot). Now, according to the PCM principle, any query in the first quadrant of q_e (shown as the grey region) cannot have lower costs than the cost of q_e. Also our experience, as borne out by extensive experimentation on commercial strength optimizers with benchmark queries suggests, that small changes in selectivity do not cause drastic changes in cost, across selectivity space. Therefore, the optimal plan cost of a query point q_p which is in the first quadrant of q_e, and not very far away from q_e (shown as a green dot) can act as a very efficient pilot-cost, while searching the optimal plan for q_e. Depending on this fact, our Pilot-Passing based diagram generation works in the following way:

We start generating the plan diagram from top-right corner in reverse row-major order. During the optimization of any point, we check if there is any previously optimized point

present in the first quadrant. If such a point is found, its cost is sent to the optimizer as the pilot-cost. We stop when all the points are optimized.

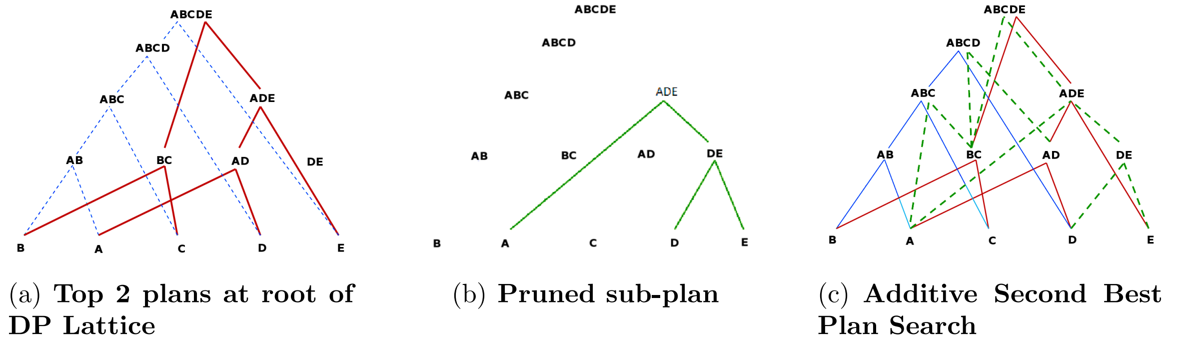
Pilot Selection. Specifically, from the first quadrant of q_e, we choose the point which is at distance of 1 from q_e along each dimension (At border cases we consider only the valid dimensions). Figure 3.4(b) shows several examples of query points (blue dots) and their corresponding pilots (green dots) in a 2D selectivity space with resolution of 5 along each dimension. Note that the top-right point (red dot) does not have a pilot and its optimization has to be carried out normally.

3.3 The PlanFill Algorithm

The PlanFill algorithm, our third and final intrusive technique is an inference based algorithm which uses the PRL and FPC features incorporated in class III optimizers. Specifically, it assumes that on each invocation, the optimizer returns both best and second-best plans (i.e. PRL with $k = 2$). Unlike the first two techniques, PlanFill achieves speedup by reducing total number of optimizations, carried out to produce the plan diagram. Also, this technique is the most instrusive in nature and significantly modify the normal course of DP. We now take a detour to present the strategies incorporated for (a) generating second-best plan (PRL) and (b) costing any plan at any point (FPC), in the optimizer kernel.

3.3.1 Generating Second-Best Plans

Modern query optimizers are based on the dynamic programming (DP) based search described in [13], to obtain the cost-optimal plan for a query. Given a query associated with multiple relations, the query optimizer searches for the best query execution plan by finding the best join strategy for successively larger subsets of relations [13]. Further, the 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 cheapest interesting order [13] are pruned. After termination of the search process, the least cost plan from the root of



(a) **Top 2 plans at root of DP Lattice** (b) **Pruned sub-plan** (c) **Additive Second Best Plan Search**

Figure 3.5: Example of DP Lattice

this search tree is chosen. Generating the second-best plan 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 and choose the second-best from this list. Unfortunately, this will not work as some plans may get *pruned early in the search process while being compared to the best plan* and may not appear in the final root node of the search tree. As an example, consider Figure 3.5(a) which 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 here. A possible candidate of second-best plan e.g. the sub-plan shown in Figure 3.5(b) got pruned earlier. As can be seen from this figure, the node ADE pruned the join order $A \bowtie DE$ as compared to $AD \bowtie E$ (best plan) for being costlier at this sub-plan level.

We now propose the strategy of generating second-best plan by augmenting the standard DP procedure.

Additive Second Best Plan Search. In this strategy, we *expand* the candidate set of sub-plans at each node of the DP lattice. The expanded set contains the second-best strategy for generating the sub-plan represented by that node in addition to the optimizer's default choice. Both these sub-plans are propagated to the higher level of the DP lattice. As shown in Figure 3.5(c), the dotted lines depict the path of the second-best sub-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 retained. In this expansion process, the true second-best

plan will be explicitly found at the root node. One issue with this approach is that it would now require *four* times the optimization overhead at each node for enumerating all possible combinations among the pairs of sub-plans arriving from lower level nodes. For example, at node $ABCDE$, it would require to 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 $(B \bowtie C)$.

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 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-plans generating them. Therefore, only one optimization, say $(AD \bowtie E) \bowtie BC$, needs to be carried out, and the results can be reused for the other three pairs. But a caution: This method of inheriting cost of best-pair cannot be applied directly to sub-plans with different interesting orders. In case, we encounter sub-plans having different interesting orders, at least one from each different interesting orders have to be costed explicitly. But practically for TPC-H templates, we have observed that during DP procedure the number of sub-plans having interesting orders is much lesser compared to the number of sub-plans without any interesting orders.

3.3.2 Foreign Plan Costing

A plan tree is costed by optimizer in a bottom-up procedure. A leaf node, which corresponds to a base relation is costed according to the estimated number of rows (cardinality), that can participate in the query from that relation. Any intermediate node is costed depending on the estimated number of rows of its children (or child in case of a unary node). Therefore, the cost of a plan tree is primarily governed by the cardinality estimates of the base relations.

Now, as per the discussion in Section 1, each query in the selectivity space is uniquely identified by the associated base relation selectivities. Therefore, given a plan tree for a particular query location q_1, if we simply change the cardinality estimates of base relations

PlanFill (QueryTemplate QT)

1. Let A be an empty plan diagram.
2. Set $q = (x_{min}, y_{min})$
3. while $(q \neq null)$
   (a) Optimize query template QT at point q.
   (b) Let p_1 and p_2 be the optimal and second-best plan at q, respectively.
   (c) for all points q' in the first quadrant of q
       if $(c_1(q') \leq c_2(q))$, assign plan p_1 to q'
   (d) Set q = next unassigned query point in A
4. Return A

Figure 3.6: The PlanFill Algorithm

(leaf nodes), we will obtain a plan for a different query q_2 in the same selectivity space. Note that, this plan may not be optimal at the new query location but if we carry out the costing procedure on this modified plan, we will eventually get the cost of that plan at this new location. This is supported by the fact that, primary inputs of plan-costing are the base relation cardinality estimates.

Therefore our strategy for costing a plan P at a foreign query location q_f is as follows:

1. Get the base relation selectivities associated with the query q_f.

2. In the plan tree of P, visit the appropriate leaf nodes and inject the constants corresponding to the new selectivities into those nodes, by modifying the associated restriction clauses.

3. Cost this modified plan tree through the usual costing process of optimizer.

We now present the PlanFill algorithm which uses the second-best plan and the foreign plan costing as APIs present in the optimizer and produces the plan diagram in a manner similar to the well-known "flood-fill" [15] algorithm.

3.3.3 The PlanFill Algorithm

The PlanFill algorithm for a 2D query template is shown in Figure 3.6. 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 best 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 with the increasing selectivity, throughout the selectivity space, which is true in practice for most query templates on all industrial-strength query optimizers [7].

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 2 *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. ■

Limitation of PCM Assumption

The PlanFill and Pilot-Passing techniques of generating optimizer diagrams, rely on the assumption of PCM and this assumption holds true for majority of the benchmark queries on commercial strength optimizers. But, in few cases we have also observed optimizer diagrams, violating the PCM principle [7]. In such a situation, the PlanFill algorithm will produce erroneous optimizer diagrams, as the Theorem 2 does not hold true any more. Even worse, the Pilot-Passing technique will fail to produce a complete plan for some of the queries. Specifically, where pilot-cost is lesser than the optimal plan cost, the entire search space will be pruned and an empty candidate set will be produced at the final level of DP (we need to resort to a fresh optimization from scratch, if such a situation arises). The Subplan-Caching technique however is independent of any prior assumption about the optimizer's behavior.

Cost and Cardinality Diagrams

The cost and cardinality diagrams for class III algorithms are automatically generated during plan diagram generation process. The PlanFill algorithm generates perfect cost and cardinality diagram, as every point in the selectivity space is either optimized or costed using FPC. The Pilot-Passing and Subplan-Caching techniques optimize every point in the selectivity space and therefore, produce perfect cost and cardinality diagrams.

Chapter 4

Implementation Details

All of the algorithms described in Section 2 and 3 are implemented in the Picasso database query optimizer visualizer tool [17] and PostgreSQL [16]. We present a rough summary of implementation details in this chapter.

4.1 Picasso

Picasso query optimizer visualizer facilitates the generation of optimizer diagrams. The non-intrusive approximation techniques are implemented in Picasso by modifying the diagram generation process according to our proposed strategies. The intrusive techniques also need modification of the diagram generation process. As an example, to utilize Pilot-Passing, the diagrams need to be generated in reverse-row-major order, and to utilize Subplan-Caching, all diagonal points need to be optimized prior to any other point etc.

4.2 PostgreSQL

We have used PostgreSQL 8.3.7 [16] optimizer kernel to implement our intrusive strategies. The four major components of our implementation are described below:

Subplan-Caching. Subplan-Caching can be implemented in the optimizer kernel, with a few changes, as described below:

1. The data structure of a DP node needs to be augmented with a flag to indicate whether the node is dependent, independent or partially dependent.

2. A global array structure is needed to hold the cached sub-plans.

3. The location of the query in the selectivity space needs to be provided to the optimizer.

In our implementation, all these took around 200 additional lines of code in the PostgreSQL optimizer kernel and the global array to save sub-plans, incurs an additional memory overhead of a few kilobytes.

Pilot-Passing. Implementing the Pilot-Passing technique (as described in section 3.2.1) requires modest changes in a optimizer kernel. Though our implementation in PostgreSQL took only around 100 lines of code, Pilot-Passing alters the course of DP more than the Subplan-Caching in a logical sense. The pruning is carried out in the following way: Suppose two nodes are being considered for join, in DP. If any of them has a cost higher than the pilot-cost, we can reject that particular join immediately. This saves the time of enumerating and costing all possible join methods for those two nodes.

Second Best Plan. We have implemented the additive strategy of generating the second best plan (as described in Section 3.3.1) in PostgreSQL. The implementation includes:

1. Augmenting the data structure of a node in DP to store the second best sub-plan.

2. Enumerating all combinations between two pairs of sub-plans, through cost inheritance, at every non-leaf node of DP.

We have observed that the increased space required for expanding the candidate subplans went up to maximum $10MB$ (the original DP takes roughly $6MB$), which is quite reasonable with the current resource-rich systems. The additional time overhead for retrieving second-best plan is observed to be roughly *few milliseconds* more than the standard optimization. The coding effort required was adding or modifying around 200 lines of code.

Foreign Plan Costing. Implementing FPC in PostgreSQL was not straight-forward, because of the following behavior of the optimizer:

1. PostgreSQL maintains *Path* structure for all of the nodes in DP to hold candidate sub-plan information. But from the join root of DP to the plan root (i.e. in the plan stem), a less complex structure *Plan* is used to hold the final plan. During this conversion, a lot of useful DP related information is discarded, which is required to cost the plan later.

2. PostgreSQL caches certain temporary results during optimization and reuses them. These cached values are discarded on completion of an optimization but they are required if we want to cost a plan later.

Because of these issues, the entire costing module (roughly 5K lines of code) needed to be replicated and certain additional data structures were added to store information from *Path* structure and to carry around the cached values during a run of FPC.

Apart from these four major components, we also modified the parser module of PostgreSQL, for passing additional information like pilot-cost, query location etc. to the optimizer. For example, instead of using normal $explain < query >$, we use $explain\ pilot(< pilot - cost >) < query >$ and $explain\ subplan(< cache/reuse >, < relation >, < position >, ...) < query >$ for invoking the Pilot-Passing and Subplan-Caching respectively.

Chapter 5

Experimental Results

The testbed used in our experiments is the Picasso optimizer visualization tool [17]. The experiments were conducted over optimizer diagrams produced from a variety of two, three, and four-dimensional **TPC-H** based query templates. QTx refers to a query template based on Query x of the TPC-H benchmark. The optimizer diagrams were generated for a commercial optimizer anonymously referred to as OptCom, and the public-domain optimizer PostgreSQL. We present only a representative set of results in this thesis.

Table 5.1 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 $\theta_I, \theta_L = 10\%$ 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. We see in Table 5.1 that the RS_NN algorithm requires a substantial amount of time, or equivalently, number of optimizations, to generate the approximate plan diagram (e.g. 27% optimizations required for 3D version of query template QT*9*). On the other hand, GS_PQO exhibits a much better performance (10% for the same template) – in fact, our experience has been that it needs less than 15% of the exhaustive time *across all templates.* The actual values of identity and location errors are also presented in the table, which are never materially larger than the user given error bounds, clearly highlighting the approximation ability of our algorithms.

Dimension / Resolution	Query Temp- -late	No. of Plans	Gen Time	Approximation Time Taken		Optimizations Required (%)		RS_NN Error(%)		GS_PQO Error(%)	
				RS_NN	GS_PQO	RS_NN	GS_PQO	ϵ_I	ϵ_L	ϵ_I	ϵ_L
2D: 1000	QT8	132	6 d	29 h (21%)	4.2 h (3%)	21 %	3 %	10 %	6 %	2 %	8 %
	QT16	25	16 h	2 h (10%)	9 m (1%)	10 %	1 %	8 %	6 %	8 %	10 %
	QT21	58	2.3 d	2.7 h (5%)	32 m (1%)	5 %	1 %	12 %	9 %	2 %	6 %
3D: 100	QT8	190	6.5 d	1.6 d (26%)	16 h (10%)	26 %	10 %	11 %	2 %	8 %	10 %
	QT9	404	10 d	64 h (27%)	24 h (10%)	27 %	10 %	10 %	6 %	8 %	16 %
	QT21	130	3 d	15 h (21%)	5.8 h (8%)	21 %	8 %	2 %	4 %	10 %	13 %
3D: 300	QT8	314	4 mons	–	2 d (2%)	–	2 %	–	–	–	–
4D: 30	QT8	243	5 d	23 h (19%)	15 h (12%)	19 %	12 %	12 %	9 %	4 %	9 %

Table 5.1: Approximation Efficiency for Class I optimizers with TPC-H ($\theta = 10\%$) [OptCom]

Dimension/ Resolution	Query Template	No. of Plans	Gen Time	Approximation Time Taken	Optimiza- tions(%)	Error (%)	
						ϵ_I	ϵ_L
2D: 1000	QT8	132	6 d	3.8 h (3%)	3 %	6 %	4 %
	QT16	25	16 h	9 m (1%)	1 %	8 %	5 %
	QT21	58	2 d 6 h	32 m (1%)	1 %	9 %	10 %
3D: 100	QT8	190	6 d 10 h	6.1 h (4%)	4 %	11 %	8 %
	QT9	404	10 d	21.6 h (9%)	9 %	6 %	4 %
	QT21	130	3 d	3.5 h (5%)	5 %	4 %	4 %
3D: 300	QT8	314	4 mon	2 d (2%)	2%	–	–
4D: 30	QT8	243	5 d	15 h (12%)	12 %	4 %	4 %

Table 5.2: Approximation Efficiency of GS_PQO algorithm for Class II optimizers with TPC-H ($\theta = 10\%$) [OptCom]

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. Table 5.2 shows the effort required by GS_PQO for obtaining approximate plan diagrams with $\theta_I, \theta_L = 10\%$ on the TPC-H benchmark. We see here that GS_PQO often reduces the approximation overheads by a significant fraction as compared to the corresponding numbers in Table 5.1, testifying to the utility of FPC. As an example, consider the 3D version of the query template QT8, whose overhead is brought down from 10% to 4%, when FPC is used.

For Class III optimizers the performance of Subplan-Caching technique is illustrated in Table 5.3, on the TPC-H benchmark. The figures in this table suggest that Subplan-Caching can produce perfect diagrams with less than 70% overhead for 2D query tem-

Dimension/ Resolution	Query Template	# of Plans	Exhaustive Gen Time	Subplan-Caching Time Taken
2D:1000	QT5	22	5 h 20 m	3 h (55 %)
	QT7	23	5 h 10 m	3 h 10 m (61 %)
	QT8	20	6 h 10 m	3 h 22 m (55 %)
	QT9	16	6 h 40 m	4 h 20 m (65 %)

Table 5.3: Class III (Non-PCM) : Perfect Diagram Efficiency[PostgreSQL]

Dimension/ Resolution	Query Template	# of Plans	Exhaustive Gen Time	Pilot-Passing Time Taken	PlanFill	
					Time Taken	Opt %
2D:1000	QT5	22	5 h 20 m	3 h 15 m (61 %)	4 m (1%)	0.17 %
	QT8	20	6 h 10 m	3 h 50 m (60 %)	2 h 47 m (45%)	44 %
	QT9	16	6 h 40 m	3 h 40 m (58 %)	40 m (10%)	7.4 %
3D:100	QT5	23	5 h 48 m	3 h 15 m (58 %)	13m (3%)	2.4 %
	QT8	49	5 h 58 m	3 h 16 m (55 %)	2 h 2 m (34%)	32 %
	QT9	22	6 h 45 m	3 h 35 m (58 %)	5 m (2%)	0.24 %
4D:30	QT5	37	4 h 50 m	2 h 50 m (60 %)	25 m (8%)	5.8 %
	QT8	62	4 h 30 m	2 h 30 m (56 %)	1 hr 18 m (29%)	26 %
	QT9	28	6 h 10 m	3 h 39 m (59 %)	7 m (2%)	0.7 %

Table 5.4: Class III (PCM) : Perfect Diagram Efficiency[PostgreSQL]

Algorithm	Diagram Quality	Impact on Optimizer Codebase	PCM Assumption
RS_NN	Approximate	NIL	No
GS_PQO	Approximate	NIL	No
Subplan-Caching	Perfect	Low	No
Pilot-Passing	Perfect	Low	Yes
PlanFill	Perfect	High	Yes

Table 5.5: Comparison of Algorithms

plates. The reason only 2D query templates are considered for this algorithm, is described in Section 3.1.1.

Finally, Table 5.4 shows the performances of Pilot-Passing and PlanFill (techniques which rely on PCM) on the TPC-H benchmark. We can see that Pilot-Passing technique is capable of producing perfect diagrams consistently with around 60% overhead. On the other hand, the PlanFill produces *completely accurate* diagrams with a typical overhead of less than 10%. The large time required to produce plan diagrams for query template QT*8* can be attributed to the fact, that the cost functions of optimal and second-best plan, for this query template, are very close to each other, and therefore PlanFill algorithm is unable to "fill" a large area from the first quadrant of any optimized point.

Note that it will be incorrect to assume that PlanFill is superior than Pilot-Passing, because if we consider the complexity involved in implementation, Pilot-Passing is much cheaper than PlanFill. Also, Pilot-Passing does not need FPC – an indispensable component of PlanFill, which is not provided by all commercial optimizers. Again, we can not conclude from the performance numbers, that Subplan-Caching is an inferior technique than PlanFill or Pilot-Passing, the reason being that, Subplan-Caching can be used without PCM assumption, where PlanFill will fail to produce perfect diagrams and Pilot-Passing will fail to achieve any speedup. A comparative study of all the proposed al-

gorithms, is presented in Table 5.5, showing the relative merits of an individual algorithm compared to others.

Chapter 6

Conclusion and Future Works

We have investigated in this thesis the efficient generation of optimizer diagrams, a key resource in the analysis and redesign of modern database query optimizers. 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 have also shown that, with some changes in the optimizer kernel, the diagram generation overhead can be brought down significantly.

In our future work, we would like to study if it is possible to use two or more of these techniques in conjunction, and to achieve further reduction in diagram generation overhead e.g. we can use Pilot-Passing and Subplan-Caching in our GS_PQO algorithm. We would also like to investigate whether the perfect diagram generation methods can be relaxed in any way to produce cheap and accurate approximate diagrams. A relaxed version of PlanFill is already shown in [3], where we trade error, subject to the user given bounds, to lower the diagram generation overhead.

We hope that our results will encourage the database industry to further investigate techniques like Pilot-Passing and Subplan-Caching to see if they can be utilized to aid query optimization, in any other way.

References

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

[2] A. Dey, "Efficiently Approximating Query Optimizer Diagrams", *MSc(Engg) thesis, SERC, Indian Institute of Science*, June 2009.

[3] 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.

[4] 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*, 2008. *http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2008-01.pdf*

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

[6] 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)*, September 1995.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] A. Rosenthal, U. Dayal and D. Reiner, "Speeding a Query Optimizer: The Pilot Pass Approach", *Computer Corp. of America*, unpublished note.

[13] 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.

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

[15] Flood Fill Technique, *http://en.wikipedia.org/wiki/Flood_fill*

[16] PostgreSQL Database version 8.3, *http://www.postgresql.org/docs/8.3/static/index.html*

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