Efficient Identification of Robust Plans and Efficient Generation of Plan Diagrams

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Anshuman Dutt

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

JUNE 2010

To my Parents, whose dedication to my success and continued support,
I shall always remember

Acknowledgements

I would like to thank my advisor Prof. Jayant Haritsa, for providing invaluable guidance and encouragement throughout my project work. He has been a constant source of inspiration for me throughout my stay at IISc campus.

I would also thank current and former members of DSL for helping me through technical discussions and share some memorable moments with me.

Abstract

Predicate selectivity estimates are subject to considerable run-time variation relative to their compile-time estimates, often leading to poor plan choices that cause inflated response times. To address this problem, "Robust Plans" were introduced in [7] i.e. plans that are relatively less sensitive to such selectivity errors. In [7], robust plan were identified through anorexic reduction of plan diagrams using SEER algorithm. EXPAND algorithm, later suggested in [1], is a intrusive way to identify *Robust Plans*, that substitutes, whenever possible, the optimizer's solely cost-conscious choice with an alternative choice, robust against the selectivity errors. The primary contribution of this work is a new version of EXPAND, improved on both algorithmic and implementation aspects, that brings down the time and memory overheads.

We also present a more user-friendly way of invoking the EXPAND algorithm, implemented inside the query optimizer, by providing it as an API feature.

Another orthogonal contribution of our work is implementation of three new features in the API of a public domain query optimizer i.e. "Foreign Plan Costing" (FPC), "Plan Rank List" (PRL) and "Pilot-passing" (PILOT). These features facilitate efficient generation of plan diagrams through two independent and alternative techniques i.e. Pilot-Passing (using PILOT) and PlanFill (using FPC and PRL). The effectiveness of these techniques is already proven in [13] through offline implementation of respective features. We also implemented these independent techniques in Picasso [18], as optional ways of generating plan diagrams.

In addition to PlanFill algorithm, the FPC feature also facilitates identification of robust plans in any plan diagram using the SEER algorithm [7].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern query optimizers choose their execution plans primarily on a cost-minimization basis i.e. using the classical System R [11] strategy. The development of Picasso [18] stimulated the analysis of the working of query optimizers and hence created many possibilities of improvement in query optimizers. In this work we are focusing on two independent ways of improving query optimizers. In this chapter, we will briefly introduce the need and importance of both aspects and our contribution to improve query optimizer in respective aspects. First we will introduce the primary contribution of our work i.e. efficient identification of robust plans, in section 1. Then we will discuss the additional and orthogonal contribution i.e. efficient generation of plan diagrams in section 2.

1.1 Efficient Identification of Robust Plans

1.1.1 Background

Selectivity estimates of predicates on the base relations are critical inputs to query optimizers in modeling costs of query execution plans. However it is common knowledge that, in practice, these estimates are often significantly in error with respect to the actual values encountered during query execution. Such errors arise due to a variety of reasons [14] including outdated statistics, attribute-value-independence (AVI) assumptions and coarse summaries. An adverse fallout of these errors is that they often lead to poor plan choices resulting in inflated query execution times.

Robust Plans To address this problem, an obvious approach is to improve the quality of the statistical meta-data, for which several techniques have been presented in the literature ranging from improved summary structures [2] to feedback-based adjustments [14, 5] to on-the-fly reoptimization of queries [3, 8, 9]. A complementary and conceptually different approach suggested in [1, 7] is to identify robust

plans that are relatively less sensitive to such selectivity errors. In a nutshell, to aim for resistance, rather than cure, by identifying plans that provide comparatively good performance over large regions of the selectivity space. Such plan choices are especially important for industrial workloads where global stability is as much a concern as local optimality.

Specifically, the goal is to identify plans that are (a) guaranteed to be *near-optimal* in the absence of errors and (b) likely to be comparatively *stable* in the presence of errors located across the entire selectivity space.

Foreign Plan Costing Originally, given a query instance, query optimizers finds a plan that has optimal cost for the selectivity constants given in the query. In some situations it is useful to find the cost of this plan for a different set of selectivity constants i.e. costing of given plan at some foreign location in the selectivity space. This process can be termed as "Foreign Plan Costing" abbreviated as FPC.

1.1.2 Identification of Robust Plans

There are two alternative ways to identify robust plans that are entirely different in their approach i.e 1. SEER algorithm 2. EXPAND algorithm.

SEER algorithm identifies robust plans for the whole selectivity space by first generating the plan diagram and then reducing the plan diagram. This approach is non-intrusive i.e. it treats query optimizer as a black box.

On the other hand, EXPAND algorithm gives a alternative robust plan, if possible, for given a query instance. EXPAND is an intrusive approach that require changes in optimizer's DP-routine. EXPAND [1] is based on judiciously expanding the candidate set of plan choices that are retained during the core dynamic-programming exercise based on both cost and robustness criteria rather than only cost.

1.1.3 Challenges

The SEER algorithm requires FPC feature to be present in the API of query optimizer. And the EXPAND algorithm faced the challenge of keeping the overheads, due to plan expansion, reasonably low while maintaining the plan-stability and safety performance.

1.1.4 Our Contributions

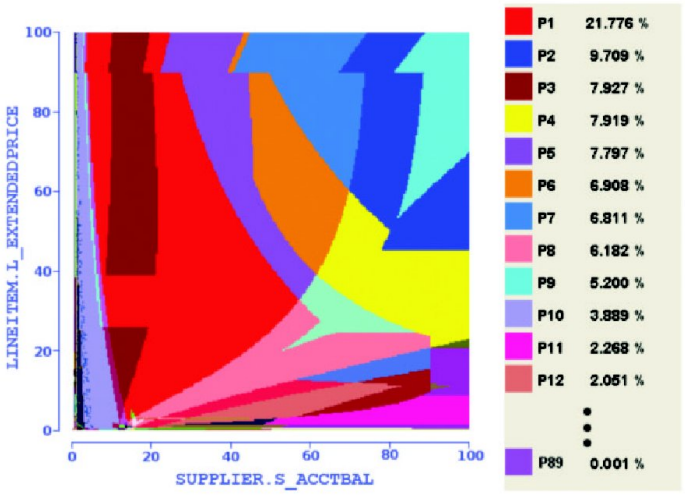
We provide a generic FPC feature, crucial for the SEER algorithm, at the API level of the optimizer. It means, now we can find the cost of the optimal plan for the given query at user-specified remote

location in the user-specified selectivity space.

EXPAND algorithm, being an intrusive technique, require changes to the plan enumeration process of the query optimizer. We present here a more efficient version of EXPAND algorithm has been improved on both algorithmic aspects like a better plan enumeration process, inheritance of foreign costs, improved C-S-B skyline check and implementation aspects like a user-friendly interface for requesting a robust plan rather than the optimal plan.

```
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 = 'ECON-
             OMY ANODIZED STEEL' and
             s_acctbal :varies and l_extendedprice :varies
        ) as all_nations
group by o_year
order by o_year
```

(a) **Query Template**



(b) **Plan Diagram**

Figure 1.1: **Example Query Template and Plan Diagram (QT8)**

1.2 Efficient Generation of Plan Diagrams

1.2.1 Background

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 [10] 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 1.1(a), based on Query 8 of the TPC-H benchmark. The associated plan diagram for QT8, on a popular commercial database engine, is shown in Figure 1.1(b).

Since their introduction in [10], plan diagrams have proved to be a powerful metaphor for the analysis and redesign of industrial-strength database query optimizers.

1.2.2 Generation of Plan Diagrams

The generation and analysis of optimizer diagrams has been facilitated by the Picasso optimizer visualization tool [18]. Given a multi-dimensional SQL query template like QT8 shown in Figure 1.1(a) and a choice of database engine, the Picasso tool produces the associated plan diagram 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.

1.2.3 Challenges

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.

1.2.4 Solution Approach

[13] suggests two alternative ways to reduce the computational overheads in the plan diagram generation process. One is to generate an approximation to the original diagram and other is to generate perfect diagrams with lower overheads. Here we will concentrate on the perfect diagram generation techniques since they need support of special features in the query optimizer for their working. Let us first take a quick look at the special features.

1. Plan Rank List: Originally, given a query instance, the query optimizer returns just the cost-optimal plan for the query. The "Plan Rank List" feature, as the name implies, is the ability of the query optimizer to return some extra plans alongwith the cost-optimal plan, ranked on the basis of their cost i.e. the TOP-k plans where 'k' is specified by the user. It will be referred in further discussion with the abbreviation PRL.

2. Foreign Plan Costing: This feature is already described in the section 1.

3. Pilot-passing: This features enables user to specify an upper bound on the cost of plans to be considered by the query optimizer to find the optimal plan.

The first technique for perfect diagram generation is the "Pilot-passing technique", which requires modifications to dynamic programming based optimization process. It speeds up an individual optimization to bring down the overall diagram generation overhead. The other technique for perfect diagram generation is PlanFill algorithm, which needs significant amount of changes in optimizer and uses features like PRL and FPC to generate plan diagrams while optimizing only a small subset of points in the selectivity space.

1.2.5 Our Contributions

The effectiveness of these techniques is already proven in [13]. The results were based on offline application of these techniques. We made the features FPC and PRL easily usable in optimizer at API level and also modified Picasso to support the efficient generation of plan diagrams using *Pilot Passing* and *PlanFill* algorithms. Although the PlanFill algorithm in [13] used only TOP-2 plans, the implementation of PRL is generalized in the sense that user can ask for TOP-K plans rather than just TOP-2 plans. Similarly, the changes in Picasso are generic to facilitate a possible extension of PlanFill algorithm to use k-best plans rather than just second-best plan.

Chapter 2

Identification of Robust Plans through EXPAND

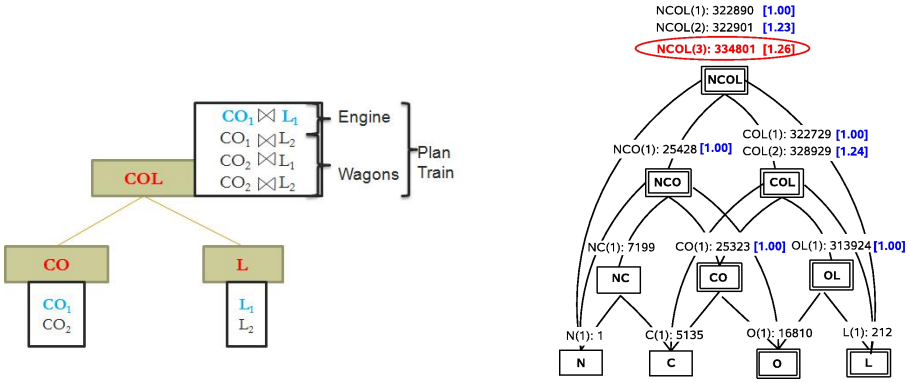
This chapter describes in detail the original EXPAND algorithm and then all the proposed improvements with a few important points about implementation.

2.1 Introduction

In the classical DP procedure[11], only the cheapest sub-plan identified at each lattice node is forwarded to the upper levels. And at the root node, the cheapest plan is returned as the optimizer's choice. This has been modified in the EXPAND family of algorithms. EXPAND is based on judiciously expanding the candidate set of sub-plan choices that are retained during the core dynamic-programming exercise, based on both cost and robustness criteria. Now, we present the generic process followed in our EXPAND family of algorithms. There are two aspects to the algorithms: First, a procedure for expanding the set of sub-plans retained in the optimization exercise, and second, a selection strategy to pick a stable replacement from among the retained sub-plans. The details can be found in [1]. For all the further discussions in the report, out of the family of EXPAND algorithms, we will focus only on the *NodeExpand*, as it was the algorithm suggested by [1].

2.2 Terms & Notations

For ease of understanding, we will use the term "train' to refer to the expanded array of sub-plans that are propagated from one node to another, with the "engine' being the cost-optimal sub-plan (i.e. the one that DP would normally have chosen) and the "wagon's' the additional sub-plans, as shown in



(a) **Plan Train** (b) **Example**

Figure 2.1: **NodeExpand with Example**

Figure 2.1(a). The notation X is used to indicate a generic node in the DP lattice.

The engine is denoted by p_e, while p_w is generically used to denote the wagons (the lower-case p indicates a sub-plan as opposed to complete plans which are identified with P).

Query locations in the selectivity space are denoted by q_i. Specifically, q_e is used to denote the local query instance and q_a is used to denote query instance at any foreign location in the selectivity space.

2.3 Plan Expansion

NodeExpand is based on judiciously expanding the candidate set of sub-plan choices that are retained during the core dynamic-programming exercise, based on both cost and robustness criteria. That is, instead of merely forwarding the cheapest sub-plan from each node in the DP lattice, a train of sub-plans is sent, with the cheapest being the engine, and viable alternative choices being the wagons. The final plan selection is made at the root of the DP lattice from amongst the set of complete plans available at this terminal node, subject to user-specified cost and stability criteria.

2.3.1 Leaves and Internal Nodes

Given a query instance q_e, at each error-sensitive leaf (i.e. base relation) or internal node x in the corresponding DP lattice, the following four-stage retention procedure is used on the set of candidate wagons generated by the standard exhaustive plan enumeration process.

1. Local Cost Check: In this first step, all wagons whose cost is more than $(1 + \lambda_l^x)$ of the engine p_e are eliminated from consideration. Here, λ_l^x is an algorithmic cost-bounding parameter that can, in principle, be set independently of λ_l, the user's local-optimality constraint (which is always applied at

the final root node, as explained later).

2. Global Safety Check: In the next step, the LiteSEER heuristic [7] is applied on all the (remaining) wagons, relative to p_e. LiteSEER is based on the "safety function"

$$f(q_a) = c(p_w, q_a) - (1 + \lambda_g^x)c(p_e, q_a) \tag{2.1}$$

which captures the difference between the costs of p_w and a λ_g^x-inflated version of p_e at location q_a. If $f(q_a) \leq 0$ throughout the selectivity space S, we are guaranteed that, were the cheapest sub-plan to be (eventually) replaced by the candidate sub-plan, the adverse impact (if any) of this replacement is bounded by λ_g^x and is, in this sense, *safe*.

Verifying the safety check is possible by exhaustively invoking the FPC function at all locations in S. However, the overheads are unviably large since the cumulative effort is proportional to the product of the number of sub-plans at the node and the number of points in S. Typical values of this product are in excess of a million, making an exhaustive approach impractical.

To address this efficiency issue, the LiteSEER heuristic simply evaluates whether all the *corners* are safe, that is,

$$\forall\ q_a \in Corners(\mathsf{S}),\ f(q_a) \leq 0 \tag{2.2}$$

The intuition here is that if a replacement is known to be safe at the corners of the selectivity space, then it is also highly likely to be safe *throughout the interior region* (the reasons for this expectation are discussed in detail in [7]).

Note that λ_g^x is also an algorithmic parameter that can be set independently of λ_g (which is always applied at the final root node, as explained later). As a practical matter, we would expect the choice to be such that $\lambda_g^x \geq \lambda_l^x$.

3. Global Benefit Check: While the safety check ensures that there is no material harm, it does not really address the issue of whether there is any *benefit* to be expected if p_e were to be (eventually) replaced by a given wagon p_w. To assess this aspect, we compute the benefit index of a wagon relative to its engine as

$$\xi(p_w, p_e) = \frac{\overline{c}(p_e, q_a)}{\overline{c}(p_w, q_a)} \qquad q_a \in Corners(\mathsf{S}) \tag{2.3}$$

That is, a CornerAvg heuristic is used wherein a comparison of the arithmetic mean of the costs at the *corners* of selectivity space S is used as an indicator of the potential assistance that will be provided throughout selectivity space S. Benefit indices greater than 1 are taken to indicate beneficial replacements whereas lower values imply superfluous replacements. Accordingly, only wagons that have $\xi > 1$ are retained and the remainder are eliminated.

4. Cost-Safety-Benefit Skyline Check: After the above three checks it is possible that there may be some wagons that are "dominated" – that is, their local cost is higher, their corner costs are individually higher, and their expected global benefit is lower – as compared to some other wagon in the candidate set. Specifically, consider a pair of wagons, p_{w1} and p_{w2}, with p_{w1} dominating p_{w2} at the current node. As these wagons move up the DP lattice, their costs and benefit indices come *closer* together, since only *additive* constants are incorporated at each level – that is, the "cost-coupling" and the "benefit-coupling" between a pair of wagons becomes *stronger* with increasing levels. However, and this is the key point, the domination property *continues to hold*, even until the root of the lattice, since the same constants are added to both wagons.

Given the above, it is sufficient to simply use a *skyline* set [4] of the wagons based on local cost, global safety and global benefit considerations. Specifically, for 2D error spaces, the skyline is comprised of five dimensions – the local cost and the four remote corner costs (the benefit dimension, when defined with the CornerAvg heuristic, becomes redundant since it is implied from the corner dimensions).

A formal proof that the above skyline-based wagon selection technique is equivalent to having retained the entire set of wagons is given in [1].

When the multi-stage pruning procedure completes, the surviving wagons are bundled together with the p_e engine, and this train is then propagated to the higher levels of the DP lattice.

2.3.2 Root Node

When the final root node of the DP lattice is reached, all the above-mentioned pruning checks (*Cost, Safety, Benefit, Skyline*) are again made, with the only difference being that both λ_l^x and λ_g^x are now *mandatorily* set equal to the user's requirements, λ_l and λ_g, respectively. On the other hand, the choice of the benefit threshold, $\delta_g (\delta_g \geq 1)$, which determines what minimum benefit is worth replacing for, is a design issue. Using a lower value and thereby going ahead with some of the stability-superfluous replacements may help to achieve *anorexic* plan diagrams [6], which is a potent objective in query optimizer construction.

2.4 Plan Selection

At the end of the expansion process, a set of complete plans are available at the root node. There are two possible scenarios:

1. The only plan remaining is the standard cost-optimal plan P_{oe}, in which case this plan is output as the final selection; or

2. In addition to the cost-optimal plan, there are a set of candidate replacement plans available that

are all expected to be more robust than P_{oe} (i.e. their $\xi > \delta_g$). To make the final plan choice from among this set, our current strategy is to simply use a *MaxBenefit* heuristic – that is, select the plan with the highest Benefit Index.[1]

An example of the whole process is shown in Figure 2.1(b).

2.5 Handling Interesting Orders

To make the algorithmic description complete, we must add that each node does not have just one train but a parallel array of sub-trains, one for each interesting-order. For the sake of uniformity, we treat the plans with no-order to be a part of generic result order called NO_ORDER.

Plan Enumeration at node X involves exhaustively combining *all* sub-trains of A with *all* sub-trains of B. Subsequently, the result order of each plan-combination is determined and thus assigned to the corresponding sub-train for the node X.

Moreover, plan retention process is done *independently* for each sub-train of the node as described for a single train.

[1] In the unlikely event of ties, they can be broken by choosing the plan with the least local cost from this set.

Chapter 3

Problem Identification

3.1 Problem Definition

The *NodeExpand* algorithm need to generate and process much more sub-plans than the normal DP-routine so that final plan is not just the minimum cost plan but also better according to robustness criteria. Hence, the *NodeExpand* algorithm may need much more time and memory than the normal DP-routine. So, the problem is to explore, find and establish solution techniques to improve the *NodeExpand* algorithm to achieve the best balance between plan stability performance and overheads (time as well as memory) for a given query and user-specified cost and safety thresholds.

3.2 Analysis of Overheads

As explained earlier, the obvious reason for the higher time and memory overheads of *NodeExpand* as compared to the original DP-process, is the generation and processing of much more sub-plans at each error-sensitive node (a node which includes an error-sensitive relation). To take a closer look at the distribution of the time overheads, we performed profiling of the query optimization process using the EXPAND algorithm whose results are summarized in the table 3.1

So, we can experimentally conclude that the main sources of time overheads are plan expansion, FPC cost computations and C-S-B skyline check. The memory overheads, on the other hand, are direct

Step	QT10	3DQT8	AI3DQT8
Plan Expansion	1.4	24.5	60.4
Foreign Costing	2.1	240.3	669.9
First 3 checks	0.12	16.7	47.8
C-S-B Skyline	0.02	724.5	4326.5

Table 3.1: Distribution of Time Overheads (in ms): EXPAND

consequence of, need to store more sub-plans at each error-sensitive node and store their foreign costs. Moreover, the number of sub-plans surviving the plan-restriction process indirectly affects the memory overheads, as more survivors at current node implies more plan-expansion at nodes at the higher level.

Chapter 4

Our Contributions to EXPAND

In this chapter, we present improvements to the EXPAND algorithm. Specifically, we have focused on the possible reasons for high overheads as shown experimentally in the table 3.1, and each proposed change in the algorithm attacks one of those reasons to bring down the overall overheads of the algorithm.

4.1 Terms & Notations

Here we use, an extended set of notations in which, we provide more detail about each plan in the notation itself. We shall use these notations to make the text concise and easier understanding of the concept.

Specifically, the engine, for node X, with order o_i is denoted by $e_X^{o_i}$, while $w_X^{o_i}$ is generically used to denote the wagons for node X with order o_i. Unordered engine and wagons for node X are denoted by e_X^{no} and w_X^{no} respectively. A generic sub-plan, engine or wagon, for node X can be denoted by $p_X^{o_i}$ (the lower-case e, w and p indicates a sub-plan as opposed to complete plans which are identified with E, W and P).

Moreover, we use the term $p_X^{o_k}$*(*p_A*,*p_B*,jointype)* for the candidate-subplan of order o_k for node X, formed by joining p_A and p_B, and the way of joining used is *jointype*.

4.2 Restricted Plan Expansion

Plan Expansion is the first step of the EXPAND algorithm and is observed to be one of the major reasons of time and memory overheads. It is necessary to do plan expansion, because we need to consider extra wagons at each node other than the engine, to find out a more robust and safe choice. Moreover, we need to consider all such possible choices at all internal nodes, since we cannot predict which of them will be most robust at the top-most i.e. Root node. The creation and costing of one wagon is not a costly

Plans	QT10	3DQT8	AI3DQT8
Pruned LCC_1	138	900	9060
Pruned LCC_2	83	641	2488
Pruned LCC_3	103	1315	4849
Accepted	93	982	2024

Table 4.1: Impact of Restricted Plan Expansion

operation in itself, but the exhaustive enumeration all possible wagons i.e. combining all subtrains of A with all subtrains of B is a quite costly task. We need some way to restrict the plan enumeration process so that we can avoid creation and costing of some wagons, if possible. Moreover, we can reject some more wagons just after their creation so that plan restriction (which include more reasons of overheads) need to applied on lesser number of wagons.

4.2.1 Motivation

The main idea is to apply Local-cost check dynamically during plan enumeration, to avoid the creation of some wagons and then rejection of some more wagons just after their creation and costing. The effectiveness of the idea is well supported by the experiments summarized in Table 4.1, which shows the pruning ability of each of the following ideas.

The creation of any wagon $p_X^{o_i}$ can be avoided if we know, in advance, that it will have $(1 + \lambda_{l^x})$ times more cost than the corresponding order engine $e_X^{o_i}$. For this to happen, there are two possibilities, stated as follows:

1. the $p_X^{o_k}$ *(*$w_A^{o_i}$*,*$w_B^{o_j}$*,jointype)*, can be avoided (from generation process), if the $p_X^{o_k}$ *(*$e_A^{o_i}$*,*$e_B^{o_j}$*,jointype)* is known to be $(1 + \lambda_{l^x})$ times more costly than engine $e_X^{o_k}$. It means, a wagon-wagon combination can't pass the local cost check if the corresponding engine-engine combination has failed the local cost check. This type of pruning is being referred as LCC_1.

2. at least one of the sub-plans p_A or p_B, that will form $p_X^{o_i}$ after combination, is $(1 + \lambda_{l^x})$ times more costly engine $e_X^{o_i}$. This can happen in case that $e_X^{o_i}$ was generated from some other combination of A and B i.e. some other join order. This type of pruning is being referred as LCC_2.

Similarly, a wagon $p_X^{o_i}$ which after creation, is found to have $(1 + \lambda_{l^x})$ times more cost than the corresponding order engine $e_X^{o_i}$, can be rejected instantaneously.This type of pruning is being referred as LCC_3.

4.2.2 Challenges

The idea explained above is not straightforward to use. First of all, we need to know the engine for each train of node X, i.e. $e_X^{o_i}$. Moreover, we need to know, for each wagon-wagon combination

$p_X^{o_k}$*(*$w_A^{o_i}$*,* $w_B^{o_j}$*,jointype)*, that whether the corresponding engine-engine combination $p_X^{o_k}$*(*$e_A^{o_i}$*,* $e_B^{o_j}$*,jointype)* has passed the local cost check or not.

4.2.3 Solution

The identification of engine for each sub-train is also done by the normal DP-routine. So, we can use the normal DP-routine to first identify all engines for the node and then perform plan enumeration that is restricted using above specified three possibilities. Hence, for each node, restricted plan enumeration is done in two passes. In the first pass, normal DP-routine is followed, in which all engines for subnode A i.e. $e_A^{o_i}$ are combined with all engines for subnode B, $e_B^{o_i}$, in all possible ways. For each such sub-plan combination p_{AB}, the resulting interesting order o_i is determined and $p_{AB}^{o_i}$ is added to the plan-train-i of node X, only if it is cheaper than some already existing $p_X^{o_i}$, otherwise, it is discarded. In this way, only engines for each plan-train, $e_X^{o_i}$, are remembered for the node x.

In the second pass, for each combination of $e_A^{o_i}$ and $e_B^{o_j}$, with a specific jointype, first it is determined whether it passes the local cost check or not. (the process is explained in section 4.3.4 to avoid restatement of a similar idea) If this combination passes the local cost check, only then $w_A^{o_i}$ and $w_B^{o_j}$ combinations are created, using the specific jointype, otherwise we skip the remainder of train-i (subnode A) and train-j (subnode B), for this jointype.

The concept that lies behind this approach is that - all sub-plans in any train produce same amount of data with same interesting order. Hence, cost of *joining* any pair of sub-plans from two trains of subnodes (train-i for A and train-j for B) must be same, for a specific jointype i.e. *cost(*$w_A^{o_i}$*,* $w_B^{o_j}$*,jointype)* = *cost(*$e_A^{o_i}$*,* $e_B^{o_j}$*,jointype)* Now, out of all such combinations, the sub-plan formed by the pair of engines will have minimum total cost.

As explained earlier, we can also avoid creation of $p_{AB}^{o_i}$ if at least one of the sub-plans i.e. p_A or p_B, is $(1 + \lambda_{l^x})$ times more costly than engine $e_X^{o_i}$. But, since it is not possible to tell the result order of p_{AB} without its creation, so this idea seems unusable in theory. We have found a practical way around this, instead of comparing with the engine of same interesting-order, we can compare the cost of p_A and p_B with cost of engine that is costliest among all engines i.e. $\max(e_X^{o_i})$ for all o_i. It is obvious that, if any of p_A or p_B has cost $(1 + \lambda_{l^x})$ times more than that of the $\max(e_X^{o_i})$, then the cost of resulting p_{AB} will surely be more than $(1 + \lambda_{l^x})$ times the cost of engine e_X with same interesting-order.

It is important to note that, if the engine-i,engine-j sub-plan combination passes the local cost check, even then some of the wagon-i, wagon-j sub-plan combinations may fail this check. But, such cases can not be determined unless we create and cost that sub-plan combination. For each such wagon $p_X^{o_i}$ we check whether it has $(1 + \lambda_{l^x})$ times more cost than the corresponding order engine $e_X^{o_i}$. If yes, it is rejected otherwise passed to the next step of the algorithm.

4.2.4 Implementation

To implement this solution approach, we need to store, in each sub-plan, an extra variable which can be used to differentiate a engine from a wagon. The variable can be set accordingly after all wagons are added to the plan-train. We also need an extra variable in each sub-plan, which can help us to identify its interesting-order i.e. train number. Both of these can be done in one scan of the plan-train, in which engine is identified and the order of each wagon is saved in the wagon.

4.3 Efficient Computation of Foreign Costs

After the restricted plan enumeration and early local cost pruning at any node, we have a list of plan-trains, one for each interesting-order. Now, to apply later checks for further pruning of wagons, we need to determine, for each wagon as well as engine, foreign costs at all corners of the selectivity-space. As explained earlier, it takes considerable time to cost each sub-plan at each corner. So, we have found a way to efficiently compute the foreign costs. The efficiency is achieved in two ways. Firstly, the number of wagons for which foreign costs are computed is reduced by computing foreign costs only for local cost check survivors. Secondly, foreign costs are not *computed* for each wagon, but rather *inherited*, whenever possible, as explained below.

4.3.1 Inheritance of Foreign Costs

We can avoid computing foreign costs for some sub-plans and actually inherit them from already computed foreign costs of some specific sub-plans. The inheritance of costs is already suggested in EXPAND, we just implemented the idea restated below.

When two plan-trains arrive and are combined at a node, the costs of combining the engines of the two trains in a particular method is exactly the same cost as that of combining any other pair from the two trains. This is because the engines and wagons in any train all represent the same input data. There, any wagon-wagon combination can inherit cost of joining from the corresponding engine-engine combination. Inheritance of foreign costs can help to great extent in reducing total time overheads, as the proportion of time spent on it is quite large, as shown in Table 4.1. The effectiveness of the idea is quite clear from results in Table 4.2, that gives a comparison of number wagons for which foreign costs are computed/inherited.

4.3.2 Challenges

The idea of inheritance says that we can reuse the costs of joining two engines, for a specific jointype, i.e. $p_X^{o_k}(e_A^{o_i}, e_B^{o_j}, jointype)$ to evaluate the costs of joining any two wagons of the same respective interesting-order i.e. $p_X^{o_k}(w_A^{o_i}, w_B^{o_j}, jointype)$ But, the challenge lies in identifying $p_X^{o_k}(e_A^{o_i}, e_B^{o_j}, jointype)$, out of all

Num. of Wagons	QT10	3DQT8	AI3DQT8
Computed FCs	72	880	698
Inherited FCs	21	1347	1814

Table 4.2: Effectiveness of Inheritance of Foreign Costs

already generated sub-plans $p_X^{o_k}$.

This is not at all a trivial task as it requires tracing the train with interesting order o_k to find wagons with same 'jointype'. Then, for each such match, check whether the respective sub-plans are of interesting orders o_i and o_j. Given that comparing the interesting order for a pair of plans is itself a costly operation the whole process is a costly way of identifying $p_X^{o_k}$*(*$e_A^{o_i}$*,* $e_B^{o_j}$*,jointype)*.

4.3.3 Solution

The inheritance of foreign costs is achieved by joining two subnodes A and B (to give node X) for a specified *jointype* at a time and during this process the result of local cost check and join cost for each pair engines is remembered in separate 2D arrays indexed by (i,j) denoting 'i^{th}' & 'j^{th}' trains for subnodes A and B respectively. To use the join costs for any pair of wagons from train-i and train-j respectively, the correct pair of join costs is found by accessing the 2D array at index (i,j). This process is repeated for different ways of joining for the same subnodes A and B. And then, the whole process is repeated for all other valid subnodes in place of A and B, respectively, which can join to give node X.

In a similar fashion, the result of local cost check is used, as explained in section 4.2.3, to decide whether to consider wagon-wagon combination for this pair of engines.

4.3.4 Implementation

To implement the above described solution, we need an extra variable in each sub-plan which is used to store the information which can be used to identify, to which plan-train the sub-plan belongs to. Remember, this is also being used restricting the plan-enumeration, but there we needed to use the interesting order of the resulting subplan to perform local-cost check, whereas in this case, we are using the interesting-orders of sub-plans p_A and p_B to inherit the join-cost.

4.4 Improved C-S-B Skyline Check

The C-S-B skyline check compares each pair of wagons surviving the local cost, global safety and global benefit checks to find whether one of them is dominating the other one. Now, since the number of such comparisons is proportional to square of the number of wagons in the plan-train. Moreover, it applied in the same way to wagons of each plan-train of different interesting order. It can be a major source of time overheads, as found in experiments, especially for the nodes at higher levels in the DP-lattice.

The improvement in C-S-B skyline check is achieved in two ways. Firstly, the number of comparisons is decreased by using the full power of the concept of domination. Secondly, the definition of domination is relaxed a little bit to allow more wagons to be dominated and hence less wagons to be forwarded to the next level. Table 4.3 gives a comparison of the original and the improved CSB skyline to show the effectiveness of the idea.

4.4.1 Early Rejection in Skyline Check

The aim of CSB-skyline check is to discard redundant wagons out of whole set of competing wagons that passed the global benefit check. It does so by doing pairwise comparison between each pair of wagons. Finally, all the dominated wagons are discarded and all the dominant wagons are forwarded to the next level. After each comparison between p_1 and p_2, either one of them dominates or they are found unrelated i.e. no one dominates. A wagon is dominant iff it is not dominated by any other wagon (of the same train ofcourse). The key observation here is that since we forward only dominant wagons to the next level. So, as soon as we find that a wagon is dominated by some other wagon, it is certain that it can not be dominant. So, all the remaining comparisons for the dominated wagon are a wasted effort and hence can be avoided. This improvement may cause a large number of reduction in passes in a potential nested loop. Hence, this helps in significant saving in terms of time. It is important to note that more the number of competing wagons, more proportion of overheads are reduced.

4.4.2 Relaxed Skyline Check

4.4.3 Motivation

At any node where plan expansion is performed, we observed that, for each plan-train, among the wagons that survived the 4-stage pruning mechanism, there were many wagons that have, their local and corner costs, numerically very close to respective costs of other wagons. The idea is, small differences in estimated costs can be neglected and only one out of such wagons can be used to represent all of them.

4.4.4 Implementation

The above idea is implemented, by modifying the cost-safety-benefit-skyline check. This check is done by comparing each pair of wagons to see whether one of them dominates other.

Earlier C-S-B skyline check can be stated as follow-

A wagon p_{w_1} dominates another wagon p_{w_2}, if the local cost of p_{w_1} is less than local cost of p_{w_2}, corner costs of p_{w_1} are individually less than that of p_{w_2} and the expected global benefit of p_{w_1} is more than that of p_{w_2} (implied by the corner-costs comparisons). It can stated mathematically as follow:

# of comparisons	QT10	3DQT8	AI3DQT8
Original	26	1957232	13012950
Early Rejection	22	80682	3271167
Early Rejection & Relaxed Comparisons	16	2342	3652

Table 4.3: Effectiveness of Improvements in C-S-B skyline check

$c(p_{w_1},q_e)$ - $c(p_{w_2},q_e) \leq 0$
$c(p_{w_1},q_a)$ - $c(p_{w_2},q_a) \leq 0 \quad q_a \in \text{Corners}(\mathsf{S})$

After the comparison, either the dominated wagon is pruned, otherwise the pair is said to be "un-related" and the comparison does not have any effect. Thus, concept of domination was based on strict comparison of costs.

For implementing the Relaxed-Skyline, for any pair of wagons, at first, strict-comparison check is tried as described above. If the strict comparison declares them are *un-related*, then, the comparison of wagons are relaxed i.e. for each cost comparison, instead of strict comparison, it is checked whether the larger cost lies within a small predefined *eqv_range* with respect to the smaller cost. If it happens for each and every cost comparison, for the given pair of wagons, the compared wagons are considered to be *skyline-equivalent*. After determining skyline-equivalence between a pair of wagons, the wagon with better global benefit index is retained as dominant. The Relaxed-skyline comparison can be stated mathematically as follow:

$| c(p_{w_1},q_e) - c(p_{w_2},q_e) | \leq \text{eq\_range}$
$| c(p_{w_1},q_a) - c(p_{w_2},q_a) | \leq \text{eq\_range} \quad q_a \in \text{Corners}(\mathsf{S})$
$\xi\ (p_{w_1},p_e) \geq \xi\ (p_{w_2},p_e)$

The idea is, small differences in estimated costs can be neglected. Any other criteria could be used to decide the dominance, our choice is simply based on maximum potential robustness, indicated by the Global benefit index.

4.4.5 Possible Side-effect

We need to make sure that, relaxing the comparison of wagons in the skyline check, does not have side-effect that are unacceptable.

One possible side-effect may be incorrect transitive domination of wagons. Since, the skyline-dominance (with strict comparisons) is transitive, i.e. if p_1 dominates p_2 and p_2 dominates p_3, then p_1 also dominates p_3, the risk of incorrect transitive domination was not there in EXPAND skyline check.

But this property does not hold for Relaxed-skyline, as skyline-equivalence itself is not a transitive relation. Hence we have taken care that as soon as a wagon is declared dominated/dominant, it cannot dominate/dominated by any other wagon. So that, there is no transitive domination at all.

One other possible side-effect could be the loss of anorexia in the corresponding plan diagrams, as different representative could be chosen out of same set of wagons at the root node, for nearby query points in the selectivity space. This will not happen for our choice of representative, since Global Benefit Index, is a global phenomenon. So, if the set of competing plans is same then we will choose same representative always. We have verified in the experimental study that anorexia of plan diagrams is maintained.

4.5 Use of CC-SEER

In EXPAND [1], it is suggested that the global safety guarantee in the algorithm can be achieved with a high cost of computational overheads, both in terms of time and memory. This is so, because CC-SEER algorithm[12] needs foreign cost of each wagon at more number of corners i.e. 4^n as compared to 2^n for LiteSEER, where 'n' is the number of error-sensitive relations in the query template. The space overheads are also more because the wagons need to carry more costs to the higher levels.

If we compare the properties of LiteSEER and CC-SEER, no doubt that CC-SEER gives us safety guarantee which LiteSEER does not give. But, there is one more important feature where LiteSEER and CC-SEER differ. That is, LiteSEER suffers from *false positives* but no false negatives, whereas CC-SEER suffers from *false negatives* but no false positives. The implication of this difference is explained below.

LiteSEER suffers from false positives means that, it can wrongly predict some actually unsafe wagons to be safe and thus allows more wagons to pass the global safety check. At lower levels of DP-routine, it can cause more overheads and at the root level it can allow a wagon to be robust choice plan, which is not actually safe according to the user specified safety constraint.

CC-SEER suffers from false negatives means that, it can wrongly predict a safe wagon to be unsafe. This can cause a wagon to be wrongly rejected by global safety check. While this is a nice property for the Root node, where safety guarantee is the primary aim. In case of lower level nodes, this can cause a potentially robust wagon to be wrongly rejected at lower node itself. Although it gives us safety guarantee even at sub-plan level, but costs high overheads.

Hence, the best way is to use LiteSEER for all the nodes at lower levels to ensure no robust choice wagon is wrongly rejected and use CC-SEER at the root node to ensure the global safety of final plan. In this way, we can achieve global safety guarantee with only a little extra overhead of calculating and storing extra corner costs only at the Root-node. Note that, we are sacrificing safety guarantee at

sub-plan level in this process.

4.6 Specifying Necessary Parameters

To find a robust plan, using EXPAND algorithm, we need to specify which of the relations in the query are error-sensitive, the minimum and maximum selectivity constants for each of the error-sensitive relations, which define the selectivity space and the local and global cost threshold values λ_l and λ_g.

4.6.1 Challenges

The challenge that lie in specifying error-sensitive relations is that, outside the query optimizer, any relation in the query is identified by relation-name and if necessary, an additional alias. On the other hand, the query optimizer assigns as dynamic relation-ID to each relation and use them to identify relations in the whole optimization process. This mapping from relation name to the relation-ID is specific to each query and hence it is not trivial to specify error sensitive relations.

The other issue is how to specify the minimum and maximum selectivity constants for each of the error-sensitive relations. Note that, user can supply only selectivities for corners, which is a generic way of specifying corners of selectivity space for any dimension and any query. The corresponding constants will depend on the error-sensitive relation and the restricting condition used on that relation in the current query, hence must be decided internally by the optimizer for the current query.

4.6.2 Solution

The issue of specifying error-sensitive relations is solved by modifying the mapping routine (relation name to dynamic relation-ID) of the query optimizer to remember the relation-IDs corresponding to the names of error-sensitive relations specified by the user. One more issue may happen, if the query has two relations with the same name and only one of them is specified to be error-sensitive. In such a situation, the user need to specify the correct alias nd the *aliases* used with the two relations in the query are used to identify the correct error-sensitive relation.

The other challenge was to find the proper constants for the corners of selectivity space given the selectivity values set by the user. But, to do the reverse mapping i.e. selectivity to constant, the idea used is based on the approach used by Picasso [18] to generate constants for a given query location.

The only difference is that Picasso could read histogram information easily by querying appropriate views provided by the database-engine, whereas our implementation being inside query optimizer itself could not fire similar queries. Instead the histogram information is accessed using specially provided functions in the query optimizer code.

4.7 Application Interface

To get a robust choice of plan for any query, we need to first set the selectivities to be used as the corners of the selectivity space. These can be set by using -

‘‘SET minCornerSelectivity = c1’’ and

‘‘SET maxCornerSelectivity = c2’’.

Then, to get a robust plan for a query QUERY, we use a optional argument in the explain query i.e.

EXPLAIN

ROBUSTPLAN(λ_l,λ_g,R_1, R_2,..)

QUERY

where λ_l, λ_g are user-specific values to be used as local and global cost threshold, and each R_i is the name of a relation which is to be considered as error-sensitive, to define the selectivity space.

Chapter 5

Support for SEER Algorithm

For any database-engine to support the offline identification of robust plans for a query template, through the application of SEER algorithm to the corresponding plan diagram, the database-engine need to support the costing of any POSP plan at some remote location in the selectivity space. The concept behind costing a plan at given remote location explained below.

5.1 Foreign Plan Costing-Process

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.
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 as supplied by the user.
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.

5.2 Application Interface

To get the optimal plan of the specified query costed at some remote location, we need to specify the the error-sensitive relations and the selectivity constants for remote location as follows:
`EXPLAIN REMOTE_FPC (`R_1`,`R_2`,...) (`C_1`,`C_2`,...) QUERY`

where each R_i is the name of an error-sensitive relation and C_i is the remote location constant for the corresponding relation.

Chapter 6

Efficient Generation of Plan Diagrams

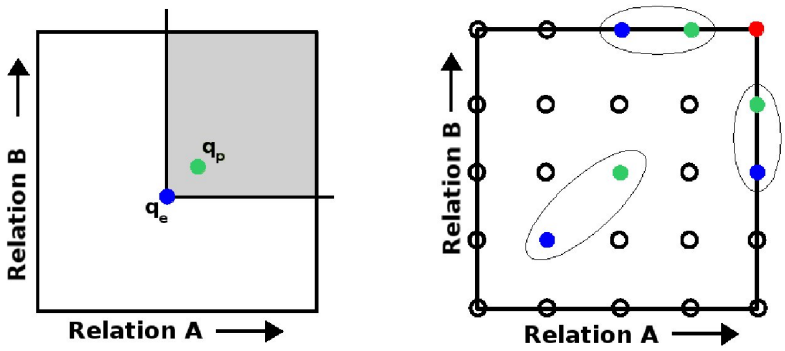
In this chapter, we present two independent ways to efficiently generate the plan diagrams. The two techniques serves the same purpose of efficient prefect diagram generation but are quite different in their approach and alternative to each other i.e. the techniques cannot be used in compliment to each other.

6.1 Pilot-Based Diagram Generation

"Pilot-Based Diagram Generation" is one of the techniques to efficiently generate Plan-Diagrams. The basic idea is to try to reduce the plan space considered during the optimization process, and therefore this technique needs a few modifications to the DP based optimization process. It helps in reducing the overheads in plan diagram generation by speeding up an individual optimization process and thereby collectively reduce the diagram generation overhead. The technique relies on the Plan Cost Monotonicity assumption about the optimizer, as explained below.

The 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 (with level) in the optimization process. If we have an upper-bound on the cost of the final plan at the root of DP lattice, said to be *PILOT-COST*, 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 and thus can avoid creation of many paths at later levels. The efficiency of Pilot-Passing technique relies heavily on obtaining an a pilot-cost which is *very close* to the optimal plan's cost.

While, for an isolated optimization, it is very hard to achieve a pilot-cost close enough to optimal



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

Figure 6.1: **Pilot Based Diagram Generation**

plan's cost (almost as hard as the actual optimization), but it can obtained very cheaply during optimizer diagram generation, on the basis of Plan Cost Monotonicity assumption of the optimizer. For the current point, the minimum cost out of all points in the first quadrant is the *PILOT-COST*.

Depending on this fact, the Pilot-Passing based diagram generation works in the following way:

"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."

6.1.1 Challenges

To be able to use the above technique for efficient generation of plan-diagrams, the first challenge is to select an appropriate *PILOT-COST* and other is to be able to pass, the selected pilot-cost with the next query, to the query optimizer, so that it can be used for pruning of candidate sub-plans. Earlier, the effectiveness of the technique was proved through an offline implementation [13].

6.1.2 Our Contribution

The online implementation of the technique need changes in Picasso and the database-engine as explained below:

Pilot Selection The Pilot-Selection is to be carried out in Picasso as follows: 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 6.1(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.

Pilot-Passing Interface The Pilot-Cost can be sent to the query optimizer along with the query as follows:

```
EXPLAIN PILOT(‘‘pilot-cost’’) QUERY
```

this pilot-cost is then used inside the optimizer to prune all the candidate subpaths with cost more than pilot-cost.

6.2 PlanFill Algorithm

The PlanFill algorithm is an inference based algorithm which needs the PRL and FPC features incorporated in the database optimizer. Specifically, it assumes that on each invocation, the optimizer returns k best plans. PlanFill achieves speedup by reducing total number of optimizations, carried out to produce the plan diagram. Also, this technique is the most intrusive in nature and significantly modify the normal course of DP. The details of the algorithm are given in the report [13].

6.2.1 Changes in PostgreSQL Optimizer

Generating Plan Rank List Finding the list of k-best plans is not as straightforward as to select the k-best plans at the root node. It can be elaborated easily for an example to find the second best plan as follows:

Suppose, the root node is ABC that can be made by joining (A)(BC) or (AB)(C). And, let us assume that, the cheapest plan is the one which is found by joining A with BC using a specific way (for instance Hash Join), then second best plan can either be the second best plan to join A with BC, in some other way or the best plan to join (AB) with (C).

Root node will have the best plan to join (AB) with (C). But, since in DP-process, we store only cheapest-way of generating each node, the second best way of joining (A) with (BC) is not even found since we had only cheapest sub-plan for (A) and cheapest sub-plan for (BC). Hence, the second best plan at the root node, may or may not be the actual second best plan for the query.

We now propose the strategy of generating top-k best plans by augmenting the standard DP procedure.

“At each node, keep k-plans for each interesting order rather than keeping only the cheapest plan. Enumerating the plans in DP process in this way, will make sure that the final root node will have true k-best plans.”

Application Interface for TOP-K plans To get Top-k plans for any query, we have added an optional argument to the explain statement, as shown below:

```
EXPLAIN TOP(‘‘k’’) QUERY
```

The output will contain k-plans one after another separated by NEXT_PLAN as separator.

Foreign Plan Costing The changes done in the optimizer and the application interface for *Foreign Plan Costing* are already described in section 5.1.

6.2.2 Changes in Picasso

The Picasso needs significant change in its plan-generation process to follow the PlanFill algorithm for efficient generation of plan-diagrams.

The changes can be described briefly as follows: The plan diagram process need to have nested loop traversal of the selctivity space for the PlanFill algorithm.

Starting from the origin of the selectivity space in row-major order, for any query point q_o which is not already optimized or *filled*, do the following:

1. find top-k plans and remember them
2. try to *fill* plan (from remembered top-k plans) for any unoptimized/unfilled point q_f in the first quadrant of the current point, using FPC feature and the concept of PlanFill (described in [13])

Note that, the PlanFill algorithm used in [13] used only top-2 plans, the implementation described here is a generalized version i.e. for top-k plans.

Chapter 7

Experimental Section

The experimental study here is only for the improvements done in the EXPAND algorithm, since other contributions are for the techniques whose impact on efficiency and effectiveness has already been studied and proven in [7] and [13].

All the modifications to the EXPAND algorithm described in the chapter 4 are implemented in PostgreSQL 8.3.6 [15] operating on a Sun Ultra 24 workstation with 3 GHz processor, 8 GB of main memory, 1.2 TB of hard disk, and running Ubuntu Linux 9.10. The user-specified cost-increase thresholds in all our experiments are λ_l, $\lambda_g = 20$ percent, a practical value as per our discussions with industrial development teams, and also a value found sufficient to provide anorexic plan diagrams in popular commercial optimizers.

To assess performance over the entire selectivity space, we took recourse to parametrized *query templates* – for example, by treating the constants associated with O.totalprice and L.extendedprice in QT10 as parameters. These templates are all based on queries appearing in the **TPC-H** and **TPC-DS** benchmarks [16, 17], and cover both 2D and 3D selectivity spaces. They feature a variety of advanced SQL constructs including groupings, orderings, nested queries, aggregates etc., and the optimization process involves handling complexities such as interesting orders and stemmed operator trees. The TPC-H database contains uniformly distributed data of size 1GB and TPC-DS that of 100GB.

7.1 Performance Metrics

The performance metrics that are important to characterize and measure the performance of *NodeExpand* are:

1. **Plan Stability:** The overall effect of plan replacements on stability is measured through the AggSERF,MinSERF and MaxSERF statistics. Further, we track **REP%**, the percentage of locations where the optimizer's original choice is replaced, and **Help%**, the percentage of error instances for which replacements were able to reduce the performance gap by a substantial margin, specifically, more than two-thirds. Lastly, we also quantify the percentage of query locations where MinSERF goes below $-\lambda_g$ as **Harm%**.

2. **Plan Diagram Cardinality:** This metric tallies the number of unique plans present in the plan diagram, with cardinalities that are less than or around *ten* considered as *anorexic diagrams* [10, 6].

3. **Computational Overheads:** This metric computes the overheads incurred, with regard to both time and space, by *NodeExpand* or its variant, relative to those incurred by the standard DP procedure.

7.2 Differences from Study in EXPAND

It is also important to specify some differences from the earlier experimental study. For any query template, the peak overheads, over the selectivity space, are specified and not averaged over the specially chosen uniformly distributed set of points. Moreover, query point locations closer to extremes of the selectivity space are also taken into account while studying the overheads. It was important since the overheads increases dramatically for some QTs as we move towards the selectivity axes. The cost bounding parameters were set to very high values for the leaf nodes to allow all possible scan choices to participate in the process.

7.3 Notations

In the subsequent discussion, we use **QTx** to denote a query template based on Query x of the TPC-H benchmark. A prefix **DS** indicates that the query template belongs to TPC-DS benchmark. By default, the query template is 2D and evaluated on a PK physical design. An additional prefix of **3D** indicates that the query template is three-dimensional, while **AI** signifies an AllIndex physical design.

For brevity, we have denoted the original implementation of NodeExpand by NE-orig, and the new implementation by NE++.

7.4 Impact on Performance

It is quite important to verify that the changes does not adversely affect the performance metrics like Plan stability, safety and plan diagram cardinality. These metrics are insensitive to inheritance of

Query Template	NE-orig							NE++						
	REP %	Agg SERF	Max SERF	Help %	Min SERF	Harm %	Plans	REP %	Agg SERF	Max SERF	Help %	Min SERF	Harm %	Plans
QT5	85	0.54	1	55	0	0	3	85	0.54	1	55	0	0	3
QT8	84	0.11	1	3	0	0	3	84	0.12	1	3	0	0	3
QT10	98	0.21	1	20	-0.24	0.01	3	98	0.22	1	20	-0.24	0.01	3
AIQT5	99	0.37	1	38	0	0	7	99	0.36	1	38	0	0	7
3DQT8	69	0.18	1	18	-2.30	0.01	3	69	0.18	1	18	-2.30	0.01	3
3DQT10	99	0.39	1	71	-0.78	2.15	5	99	0.38	1	71	-0.78	2.15	5
AI3DQT8	98	0.19	1	21	-2.80	4.30	14	98	0.19	1	21	-2.80	4.30	14
AI3DQT10	99	0.13	1	19	-4.20	0.54	26	99	0.12	1	19	-4.20	0.54	26
DSQT18	58	0.48	1	49	0	0	2	58	0.48	1	49	0	0	2

Table 7.1: Plan-Stability, safety and Plan-Diagram cardinality performance

foreign costs but could be affected by the restricting the plan expansion and relaxing the CSB-skyline check. Hence, all the required experiments were also conducted to ensure that the changes to the basic algorithm does not cause any noticeable change in these performance metrics, as show in Table 7.1.

7.5 Improvement in Time Overheads

The improvement in time overheads is a combined effect of the restricted plan expansion, inheritance of foreign costs, smarter CSB-skyline check and the Relaxed-Skyline heuristic. The impact of restricted plan expansion and inheritance of costs is already discussed. The smarter CSB-skyline ensures that the CSB-skyline check completes in minimum number of wagon comparisons for a given set of competing wagons. The Relaxed-Skyline heuristic helps by reducing the number of surviving wagons and thereby decreasing the number of sub-plans generated at the next level. Table 7.2 below shows the comparison between time overheads of earlier implementation(NE-orig) with newer implementation with all the suggested improvements(NE++).

Query Template	Optimization Time (ms)		
	DP	NE-orig	NE++
QT5	5.4	50.1	44.2
QT8	6.0	6166.9	44.1
QT10	1.5	8.4	5.7
AIQT5	6.8	91.1	59.3
3DQT8	6.0	15133.4	69.9
3DQT10	1.5	8.5	6.2
AI3DQT8	7.0	5152.3	98.8
AI3DQT10	1.9	10.9	10.5
DSQT18	5.0	2788.9	93.3

Table 7.2: Improvement in Time Overheads

Query Template	Memory Overhead (MB)		
	DP	NE-orig	NE++
QT5	2.0	12.9	7.3
QT8	2.0	49.2	8.7
QT10	1.6	3.6	2.8
AIQT5	2.7	44.4	11.1
3DQT8	2.0	119.5	11.5
3DQT10	1.6	3.8	2.8
AI3DQT8	2.8	183.5	14.1
AI3DQT10	1.7	21.2	3.7
DSQT18	2.0	53.86	14.2

Table 7.3: Improvement in Peak Memory Consumption

7.6 Improvement in Memory Overheads

The improvement achieved in memory overheads is a combined effect of the restricted plan expansion and Relaxed-Skyline heuristic As explained earlier, the memory overheads are due to memory needed to store sub-plans at each level, that pass the CSB-skyline check and the fact that each plan carry extra foreign corner costs. Relaxed-skyline heuristic helps by reducing both sub-plans stored and sub-plans generated at next level, as explained earlier. In Table 7.3, we compare the memory overheads for earlier implementation (NE-orig) with newer implementation with all the suggested improvements(NE++).

Chapter 8

Conclusions and Future Work

In this work, we identified and exploited the scope of improvements in the *NodeExpand* algorithm. The improvements varied from, simply more efficient coding of already existing concepts and exploiting their full power to major changes in the algorithm, that avoid all possibilities of high time and memory overheads. We came up with the idea of restricted plan-expansion and Relaxed-Skyline, all of which together brought down the overheads significantly and implemented the already existing concept of inheritance of costs. As a orthogonal part of the work, we added three new features in the API of the query optimizer which were already proved to be useful for efficient generation of plan diagrams.

In future, we can try to find some intelligent and automatic way of defining the cost & safety thresholds (for internal nodes) that gives best deal of performance against acceptable overheads. We can also perform experiments with more query templates from benchmarks like **TPC-DS**. Moreover, we have just concentrated on the overheads study of EXPAND [1], in future, we can also try to improve the *NodeExpand* algorithm with respect to other performance metrics like Plan Stability with adversely affecting Plan safety and plan diagram cardinality.

Bibliography

[1] M. Abhirama, S. Bhaumik, A. Dey, H. Shrimal, J. Haritsa, "Stability conscious query optimization", Tech. Report. TR-2009-01, DSL/SERC, Indian Inst. of Science, July-2009, *http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2009-01.pdf*

[2] A. Aboulnaga and S. Chaudhuri, "Self-tuning Histograms: Building Histograms without Looking at Data", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1999.

[3] S. Babu, P. Bizarro and D. DeWitt, "Proactive Re-Optimization", *Proc. of ACM Sigmod Intl. Conf. on Management of Data*, June 2005.

[4] S. Borzsonyi, D. Kossmann and K. Stocker, "The Skyline Operator", *Proc. of 17th IEEE Intl. Conf. on Data Engineering (ICDE)*, April 2001.

[5] S. Chaudhuri, V. Narasayya and R. Ramamurthy, "A Pay-As-You-Go Framework for Query Execution Feedback", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2008.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] H. Shrimal, "Characterizing Plan Diagram Reduction Quality and Efficiency", Master's Thesis, Indian Inst. of Science, June 2009.
http://dsl.serc.iisc.ernet.in/publications/thesis/harsh.pdf

[13] S. Bhaumik, "Efficient Generation of Query Optimizer Diagrams", Master's Thesis, Indian Inst. of Science, June 2009. *http://dsl.serc.iisc.ernet.in/publications/thesis/sourjya.pdf*

[14] M. Stillger, G. Lohman, V. Markl and M. Kandil, "LEO, DB2's LEarning Optimizer", *Proc. of 27th VLDB Intl. Conf. on Very Large Data Bases (VLDB)*, September 2001.

[15] *http://www.postgresql.org/docs/8.3/static/release-8-3-6.html*

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

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

[18] *http://dsl.serc.iisc.ernet.in/projects/PICASSO/picasso.html*

Index