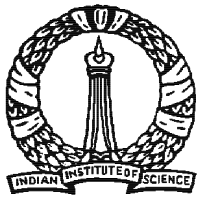
On the Stability of Plan Costs and the Costs of Plan Stability

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

By
M Abhirama



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

June 2009

Acknowledgements

I profusely thank my advisor, Prof. Jayant Haritsa, for his guidance throughout the course. Starting with day one, he has been a constant source of inspiration and motivation.

My biggest thanks goes to my family for everything they've done for me.

I thank all my friends who made my stay at IISc two of the best years of my life. I also thank my lab mates who made the time spent in lab memorable.

I also thank everyone who made my stay at the campus a pleasure.

Abstract

Modern query optimizers choose their execution plans primarily on a cost-minimization basis, assuming that the inputs to the costing process, such as relational selectivities, are accurate. However, in practice, these inputs are subject to considerable run-time variation relative to their compile-time estimates, often leading to poor plan choices that cause inflated response times.

We present in this report a parametrized family of online plan generation and selection algorithms that substitute, whenever feasible, the optimizer's solely cost-conscious choice with an alternative plan that is (a) guaranteed to be near-optimal in the absence of selectivity estimation errors, and (b) likely to deliver comparatively stable performance in the presence of arbitrary errors. The proposed algorithms have been implemented within the PostgreSQL optimizer, and their performance evaluated on a rich spectrum of TPC-H and TPC-DS-based query templates in a variety of database environments. Our experimental results indicate that it is indeed possible to identify robust plan choices that substantially curtail the adverse effects of erroneous selectivity estimates. In fact, the plan selection quality provided by our online algorithms is often competitive with those obtained through apriori knowledge of the plan search and optimality spaces. Further, the additional optimization overheads incurred by our algorithms are miniscule in comparison to the expected savings in query execution times. Finally, we also demonstrate that with appropriate parameter choices, it is feasible to directly produce anorexic plan diagrams, a potent objective in query optimizer design.

Publications

1. 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

List of Figures

List of Tables

Chapter 1

Introduction

In modern database engines, query optimizers choose their execution plans largely based on the classical System R strategy [19]: Given a user query, (i) apply a variety of heuristics to restrict the combinatorially large search space of plan alternatives to a manageable size; (ii) estimate, with a cost model and a dynamic-programming-based processing algorithm, the efficiency of each of these candidate plans;

An implicit assumption in the above approach is that the inputs to the cost model, such as selectivity estimates of predicates on the base relations, are accurate. 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 [20], 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. A variety of techniques have been presented in the literature to address the above problem, including refined summary structures [2], feedback-based adjustments [20, 7], and on-the-fly reoptimization of queries [16, 17, 4]. The particular approach we explore here is to identify, at optimization-time, *robust plans* that are relatively less sensitive to selectivity errors. In a nutshell, we "aim for resistance, rather than cure". Specifically, our 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* across the entire selectivity space. If the optimizer's standard cost-optimal

plan choice itself is robust, it is retained without substitution. Otherwise, where feasible, this choice is replaced with an alternative plan that is marginally more expensive locally but expected to provide better global performance.

Our notion of stability is the following: Given an estimated compile-time location q_e with optimal plan P_{oe}, and a run-time error location q_a with optimal plan P_{oa}, stability is measured by the extent to which the replacement plan P_{re} bridges the gap between the costs of P_{oe} and P_{oa} at q_a. Note that stability is defined relative to P_{oe}, and not in absolute comparison to P_{oa} – while the latter is obviously more desirable, achieving it appears to be only feasible by resorting to query re-optimizations and plan switching at run-time. Further, the compile-time techniques presented in this report can be used in isolation, or in synergistic conjunction with run-time approaches.

An obvious issue with regard to the plan replacement approach is whether the additional overheads involved in "second-guessing" the optimizer's default choices are adequately offset by the expected response time reductions in the presence of errors. We will demonstrate in this report, through explicit implementation within the PostgreSQL optimizer, that it is indeed feasible to achieve extremely attractive tradeoffs. Further, the run-time savings scale supra-linearly in the database size, whereas the replacement overheads are largely independent of this factor.

The EXPAND Family of Algorithms.

We propose here a family of algorithms, collectively called EXPAND, that cover a spectrum of tradeoffs between the goals of *local near-optimality*, *global stability* and *computational efficiency*. Expand 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. 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.

While the local cost information is easily obtained through the existing optimization process,

global stability is assessed through two heuristics: The first, borrowed from [12], compares, at the *corners* of the selectivity space, the costs of each wagon against the engine. The results are used to estimate whether the wagon might be *harmful* in terms of being noticeably worse than the engine with regard to global behavior. If this test is successfully passed, we bring into play the second heuristic which compares the average of the corner costs of the wagon against that of the engine to assess whether the wagon might be expected to actually *improve* the stability performance. The plan with the highest expected benefit is selected as the final choice.

Implementing the above heuristics requires the ability to cost query plans at *arbitrary* locations in the selectivity space. This feature, referred to as "Foreign Plan Costing" (FPC) in [12], is currently available in several industrial-strength optimizers, including DB2 [21] (Optimization Profile), SQL Server [24] (XML Plan) and Sybase [25] (Abstract Plan).

From the spectrum of algorithmic possibilities in the EXPAND family, we examine a few choices that cover a range of tradeoffs between the number and diversity of the expanded set of plans, and the computational overheads incurred in generating and processing these additional plans. Specifically, we consider (i) **RootExpand**, wherein the expansion is only carried out at the terminal root node of the DP lattice, representing the minimal change to the existing optimizer structure; and (ii) **NodeExpand**, wherein a limited expansion is also carried out at select internal nodes in the DP lattice. In particular, we consider an expansion subject to the same cost and stability constraints as those applied at the root node of the lattice.

To place the performance of these algorithms in perspective, we also evaluate: (i) (where feasible) **SkylineUniversal**, an extreme version of NodeExpand wherein *unlimited* expansion is undertaken at the internal nodes and the resultant wagons are filtered through a multidimensional cost-and-stability-based *skyline* [6]. The end result is that the root node of the DP lattice essentially receives the *entire plan search space*, modulo our wagon propagation heuristics. (ii) **SEER** [12], our recently-proposed *offline* algorithm for determining robust plans, wherein apriori knowledge of the parametric optimal set of plans (POSP) covering the selectivity space is utilized to make the replacements. This scheme operates from outside the optimizer, treating it as a black box that supplies plan-related information through its API.

Experimental Results. Our new online techniques have been implemented inside the Post-

greSQL optimizer kernel and their performance evaluated on a rich set of TPC-H and TPC-DS-based query templates in a variety of database environments with diverse logical and physical designs. The experimental results indicate that it is often possible to make plan choices that *substantially curtail the adverse effects of selectivity estimation errors*. Specifically, while incurring additional time overheads of the order of **10-20 milliseconds**, and memory overheads in the range of **10-100MB**, RootExpand and NodeExpand deliver plan choices that eliminate more than *two-thirds* of the performance gap for a significant number of error instances. Equally importantly, the replacement is almost *never* materially worse than the optimizer's original choice. In a nutshell, our replacement plans *"often help substantially, but never seriously hurt"* the query performance.

The robustness of our online algorithms turns out to be competitive to that of (the offline) SEER. Further, their performance is often close to that of SkylineUniversal itself. In short, RootExpand and NodeExpand are capable of achieving comparable performance to those obtained with in-depth knowledge of the plan search and optimality spaces.

Finally, while NodeExpand incurs more overheads than RootExpand, it delivers *anorexic plan diagrams* [11] in return. A plan diagram is a color-coded pictorial enumeration of the optimizer's plan choices over the selectivity space, and anorexic diagrams are gross simplifications that feature only a small number of plans without materially degrading the processing quality of any individual query. The anorexic feature, while not mandatory for stability purposes, has several database-related benefits, as enumerated in detail in [11] – for example, it enhances the feasibility of parametric query optimization (PQO) techniques [13, 14].

Another novel feature of NodeExpand is that, due to applying selection criteria at the internal levels of the plan generation process, it ensures that all the *sub-plans* of a chosen replacement are near-optimal and stable with regard to the corresponding cost-optimal sub-plan. This is in marked contrast to SEER, where only the complete plan offers such performance guarantees but the quality of the sub-plans is not assured upfront.

A valid question at this point would be whether in practice the optimizer's cost-optimal choice usually turns out to *itself* be the most robust choice as well – that is, are current industrial-strength optimizers *inherently robust*? Our experiments with PostgreSQL clearly demonstrate

that this may not be the case. Concretely, the proportion of query locations for which plan replacement took place was quite substantial – in the range of **30-50%** for providing stability, and in excess of **80%** to additionally attain anorexic plan diagrams with NodeExpand. (This observation was corroborated by results obtained on a popular commercial optimizer with SEER, where similar replacement percentages were seen.)

Finally, the plan replacement approach primarily addresses only selectivity errors that occur on the *base relations*. However, since these base errors are often the source of poor plan choices due to the multiplier effect as they progress up the plan-tree [15], minimizing their impact could be of significant value in practical environments. Further, the approach can be used in conjunction with run-time techniques such as adaptive query processing [10] for addressing selectivity errors in the higher nodes of the plan tree.

Contributions. In summary, we present a framework in this report to analyze the production of query execution plans that take into account both local-cost and global-stability perspectives. The framework opens up a rich algorithmic design space, and we explore a part of it here in the context of industrial-strength database environments. The initial results have turned out to be rather promising with regard to substantially reducing the well-known adverse impact of selectivity errors. Further, we expect that our strategies, which have been implemented in PostgreSQL as a proof-of-concept, can easily be incorporated in commercial engines as well.

To the best of our knowledge, this is the first work to investigate the online identification of stable plans that provide both guaranteed local near-optimality and enhanced global-stability in an efficient manner.

Organization. The remainder of this report is organized as follows: In Chapter 2, we describe the overall problem framework and motivation. The EXPAND approach is outlined in Chapter 3, and representative plan selection algorithms based on this approach are presented in Chapter 4. The experimental framework and performance results are highlighted in Chapter 5. Related work is reviewed in Chapter 6. Finally, in Chapter 7, we summarize our conclusions and outline future research avenues.

Chapter 2

Problem Formulation

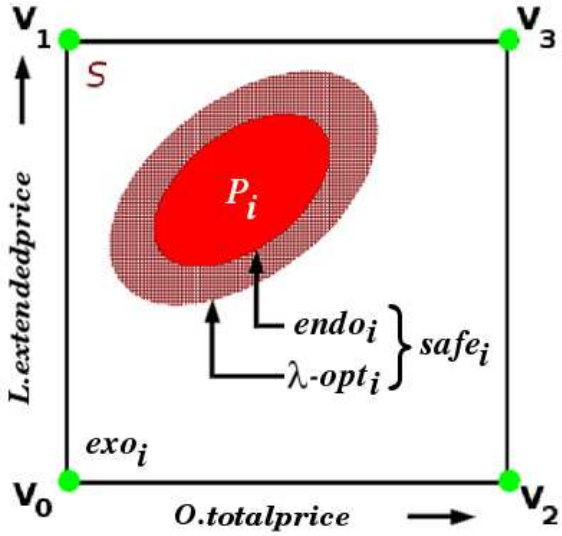
Before we begin, we would like to clarify that an implicit assumption in our study is that the query optimizer provides a reasonably accurate model of run-time performance – while we are aware that this assumption can often turn out to be off the mark in practice, improving the quality of plan cost modeling is orthogonal to the issues analyzed in this report.

Consider the situation where the user has submitted a query and desires stability with regard to selectivity errors on one or more of the base relations that feature in the query. The choice of the relations could be based on user preferences and/or the optimizer's expectation of relations on which selectivity errors could have a substantial adverse impact due to incorrect plan choices. Let there be n such "error-sensitive relations" – treating each of these relations as a dimension, we obtain an n-dimensional selectivity space S. For example, consider the sample query $\hat{Q}10$ shown in Figure 2.1(a), an SPJ version of Query 10 from the TPC-H benchmark [26] – this query has four base relations (NATION (N), CUSTOMER (C), ORDERS (O), LINEITEM (L)), two of which are deemed to be error-sensitive relations (ORDERS, LINEITEM). For this query, the associated 2D error selectivity space S is shown in Figure 2.1(b).

For ease of presentation, we will assume hereafter that S is two-dimensional (our experiments in Chapter 5 consider 3-D spaces as well). The space is represented by a finite dense grid of points wherein each point $q(x, y)$ corresponds to a query instance with selectivities x, y in the X and Y dimensions, respectively. We use $c(P_i, q)$ to represent the optimizer's estimated cost of executing a query instance q with plan P_i. The corners of the selectivity space are referred to

```
select  C.custkey, C.name, C.acctbal, N.name, C.address
from    Customer C, Orders O, Lineitem L, Nation N
where   C.custkey = O.custkey and L.orderkey = O.orderkey and
        C.nationkey = N.nationkey and
        O.totalprice < 2833 and L.extendedprice < 28520
```

(a) **Query Instance** $\hat{Q}10$



(b) **Selectivity Space**

Figure 2.1: Example Query and Selectivity Space

as V_k, with k being the binary representation of the location coordinates – e.g. the bottom-right corner $(1, 0)$, is V_2.

Given a plan P_i, the region of S in which it is optimal is referred to as its *endo-optimal* region; the region in which it is not optimal but its cost is within a factor $(1 + \lambda)$ of the optimal plan as its λ-*optimal* region (where λ is a positive constant); and the remaining space as its *exo-optimal* region. These disjoint regions together cover S and are pictorially shown in Figure 2.1(b). We will hereafter use the notation $endo_i$, λ-opt_i and exo_i to refer to these various regions associated with P_i. The endo-optimal and λ-optimal regions are collectively referred to as the plan's *SafeRegion*, denoted by $safe_i$.

2.1 Cost Constraints on Plan Replacement

Consider a specific query instance q_e, whose optimizer-estimated location in S is (x_e, y_e). Denote the cost-optimal plan choice at q_e by P_{oe}. Let the *actual* run-time location of the query be denoted by $q_a(x_a, y_a)$, and the optimal plan choice at q_a by P_{oa}.

Now, if P_{oe} were to be replaced by a more expensive plan P_{re}, clearly there is a price to be paid when there are no errors (i.e. $q_a \equiv q_e$). Further, even with errors, if it so happens that $c(P_{re}, q_a) > c(P_{oe}, q_a)$. We assume that the user is willing to accept these cost increases as long as they are *bounded* within a pre-specified local cost threshold λ_l and a global stability threshold λ_g $(\lambda_g, \lambda_l > 0)$. Specifically, the user is willing to permit replacement of P_{oe} with P_{re}, iff:

Local Constraint: At the estimated query location q_e,

$$\frac{c(P_{re}, q_e)}{c(P_{oe}, q_e)} \leq (1 + \lambda_l) \tag{2.1}$$

For example, setting $\lambda_l = 20\%$ stipulates that the local cost of a query instance subject to plan replacement is guaranteed to be within 1.2 times its original value. We will hereafter refer to this constraint as *local-optimality*.

Global Constraint: In the presence of selectivity errors,

$$\forall q_a \in \mathsf{S} \text{ such that } q_a \neq q_e, \quad \frac{c(P_{re}, q_a)}{c(P_{oe}, q_a)} \leq (1 + \lambda_g) \tag{2.2}$$

For example, setting $\lambda_g = 100\%$ stipulates that the cost of a query instance subject to plan replacement is guaranteed to be within twice its original value at all error locations in the selectivity space. We will hereafter refer to this constraint as *global-safety*.

While, in principle, λ_g can be set independently of λ_l, we expect that in practice $\lambda_g \geq \lambda_l$ would be chosen.

Essentially, the above requirements guarantee that no material harm (as perceived by the user) can arise out of the replacement, *irrespective of the selectivity error*.

2.2 Impact of Plan Replacement

Now, consider the situation where we are contemplating the decision to replace the P_{oe} choice at q_e with the P_{re} plan. The actual query point q_a can be located in any one of the following disjoint regions of P_{re} that together cover S (see Figure 2.1(b)):

Endo-optimal region of P_{re}: Here, q_a is located in $endo_{re}$, which also implies that $P_{re} \equiv P_{oa}$. Since $c(P_{re}, q_a) = c(P_{oa}, q_a)$, it follows that the cost of P_{re} at q_a, $c(P_{re}, q_a) \leq c(P_{oe}, q_a)$ (by definition of a cost-based optimizer). Therefore, improved resistance to selectivity errors is always *guaranteed* in this region. (If the replacement plan happens to not be from the POSP set, as is possible with our algorithms, $endo_{re}$ will be empty.)

λ_l-optimal region of P_{re}: Here, q_a is located in the region that could be "swallowed" by P_{re}, replacing the optimizer's cost-optimal choices without violating the local cost-bounding constraint. By virtue of the λ_l-threshold constraint, we are assured that $c(P_{re}, q_a) \leq (1 + \lambda_l)c(P_{oa}, q_a)$, and by implication that $c(P_{re}, q_a) \leq (1 + \lambda_l)c(P_{oe}, q_a)$. Now, there are two possibilities: If $c(P_{re}, q_a) < c(P_{oe}, q_a)$, then the replacement plan is again guaranteed to improve the resistance to selectivity errors. On the other hand, if $c(P_{oe}, q_a) \leq c(P_{re}, q_a) \leq (1 + \lambda_l)c(P_{oe}, q_a)$, the replacement is certain to not cause any real harm, given the small values of λ_l that we consider in this report.

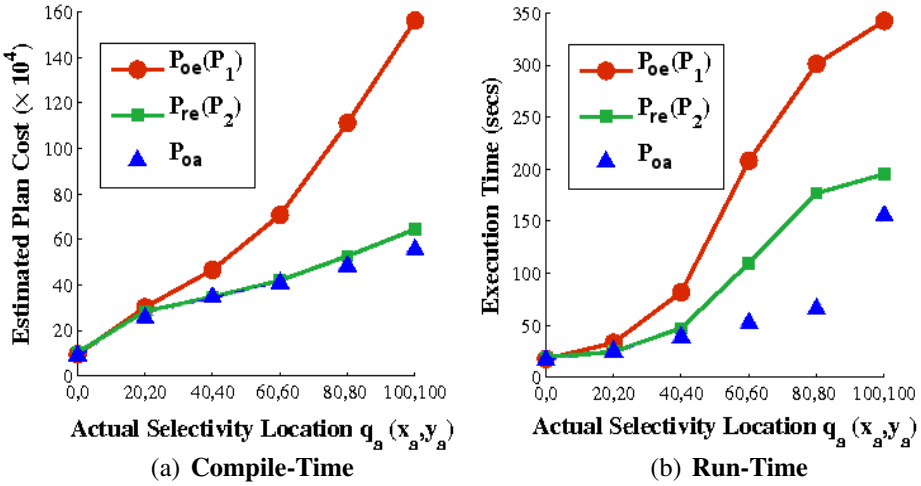
Exo-optimal region of P_{re}: Here, q_a is located outside $safe_{re}$, and at such locations, we cannot apriori predict P_{re}'s behavior relative to P_{oe}– it could range from being much better, substantially reducing the adverse impact of the selectivity error, to the other extreme of being *much worse*, making the replacement a counter-productive decision.

Explicitly establishing that the replacement of P_{oe} is not a dangerous choice anywhere in exo_{oe}, before going ahead with the substitution, is technically feasible using the FPC feature, as explained later in the report. However, it incurs unviable overheads. Therefore, we have to take recourse to heuristics instead – the silver lining is that, as shown subsequently in our experimental results, there do exist simple heuristics that are both efficient and almost invariably correct in their predictions.

2.3 Motivational Scenario

We now present a sample scenario to motivate how plan replacement could help to improve robustness to selectivity errors. Here, the example query $\hat{Q}10$ is input to the PostgreSQL optimizer; the optimizer estimates the query location q_e in S to be $(1\%, 40\%)$, its cost-optimal choice at this location is plan P_1, and the suggested replacement (by our NodeExpand algorithm with λ_l, λ_g = 20%) is plan P_2. If we plot the costs of these plans at a set of error locations q_a – for instance, along the principal diagonal of S, we obtain the graph shown in Figure 2.2(a). It is clear in this figure that the replacement plan P_2 provides substantial performance improvements on P_1. In fact, the error-resistance is to the extent that it virtually provides *"immunity"* to the error since the performance of P_2 is very close to that of the *run-time optimal* plan (generically referred to as P_{oa} in Figure 2.2(a)) at each of these locations.

To explicitly assess the above compile-time predicted performance improvements, we actually *executed* the P_1, P_2 and P_{oa} plans at these various locations – the corresponding response-time graph is shown in Figure 2.2(b). As can be seen, the broad qualitative behavior is in keeping with the optimizer's predictions, with substantial response-time improvements across the board. The somewhat decreased immunity in a few locations is attributable to weaknesses in the optimizer's cost model rather than our selection policies, and as mentioned earlier, this is an orthogonal issue that has to be tackled separately.



(a) **Compile-Time** (b) **Run-Time**

Figure 2.2: Benefits of Plan Replacement ($\hat{Q}10$, $\lambda_l, \lambda_g = 20\%$)

2.4 Error Resistance Metrics

Our quantification of the stability delivered through plan replacement is based on the **SERF** error resistance metric introduced in [12]. For a specific error instance corresponding to estimated query location q_e and cost-optimal plan P_{oe}, and a run-time location q_a, the *Selectivity Error Resistance Factor* (SERF) of a replacement P_{re} w.r.t. P_{oe} is computed as

$$SERF(q_e, q_a) = 1 - \frac{c(P_{re}, q_a) - c(P_{oa}, q_a)}{c(P_{oe}, q_a) - c(P_{oa}, q_a)} \tag{2.3}$$

Intuitively, SERF captures the *fraction of the performance gap* between P_{oe} and P_{oa} at q_a that is closed by P_{re}. In principle, SERF values can range over $(-\infty, 1]$, with the following interpretations: SERF in the range $(0, 1]$, indicates that the replacement is beneficial, with values close to 1 implying immunity to the selectivity error. For SERF in the range $[-\lambda_g, 0]$, the replacement is indifferent in that it neither helps nor hurts, while SERF values noticeably below $-\lambda_g$ highlight a harmful replacement that materially worsens the performance.

To capture the *aggregate* impact of plan replacements on improving the resistance to selectivity errors in the entire space S, we compute **AggSERF** as:[1]

$$AggSERF = \frac{\sum_{q_e \in rep(\mathsf{S})} \sum_{q_a \in exo_{oe}(\mathsf{S})} SERF(q_e, q_a)}{\sum_{q_e \in \mathsf{S}} \sum_{q_a \in exo_{oe}(\mathsf{S})} 1} \tag{2.4}$$

where $rep(\mathsf{S})$ is the set of query instances in S whose plans were replaced, and the normalization is with respect to the number of error locations that could benefit from improved robustness.

Apart from AggSERF, we also compute metrics **MinSERF** and **MaxSERF**, representing the minimum and maximum values of SERF over all replacement instances. MaxSERF values close to the upper bound of 1 indicate that some replacements provided immunity to specific instances of selectivity errors. On the other hand, large negative values for MinSERF indicate that some replacements were harmful. We measure the proportion of such harmful instances in our experiments.

An important point to note here is that while it is not possible to provide meaningful assistance in $safe_{oe}$, we still need to consider the possibility that replacements may end up causing

[1] In [12], the aggregate impact was evaluated based on the locations where replacements were made, whereas our current formulation is based on the locations where robustness is desired.

harm, reflected through negative SERF values, in these regions. This is taken into account in our calculation of MinSERF by evaluating it over the *entire* selectivity space.

2.5 Problem Definition

With the above background, our stable plan selection problem can now be more precisely stated as:

Stable Plan Selection Problem. Given a query location q_e in a selectivity space S and a (user-defined) local-optimality threshold λ_l and global-safety threshold λ_g, implement a plan replacement strategy such that:

1. $\dfrac{c(P_{re}, q_e)}{c(P_{oe}, q_e)} \leq (1 + \lambda_l)$
2. $\forall q_a \in \mathsf{S}$ s.t. $q_a \neq q_e$, $\dfrac{c(P_{re}, q_a)}{c(P_{oe}, q_a)} \leq (1 + \lambda_g)$,
   or equivalently, MinSERF $\geq -\lambda_g$.
3. The contribution to the AggSERF metric is maximized.

In the above formulation, Condition 1 guarantees local-optimality; Condition 2 assures global-safety; and Condition 3 captures the stability-improvement objective.

Chapter 3

Stable Optimization

In this section, we present the generic process followed in our EXPAND family of algorithms to address the Stable Plan Selection problem. There are two aspects to the algorithms: First, a procedure for expanding the set of plans retained in the optimization exercise, and second, a selection strategy to pick a stable replacement from among the retained plans.

For ease of presentation and due to space limitations, we will assume that there are no "interesting order" plans [19] present in the search space, and that the plan operator-trees do not have any "stems" – that is, the root join node, which represents the combination of all the base relations in the query, terminates the DP lattice. However, the algorithmic extensions for handling these complexities are described in [1], and are included in our experimental study (Chapter 5).

3.1 Plan Expansion

We now explain how the classical DP procedure, wherein only the cheapest plan identified at each lattice node is forwarded to the upper levels, is modified in our EXPAND family of algorithms – the pseudocode listing is given in Figure 3.1. 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 "wagons" the additional sub-plans. 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). Finally, the notation x is used to indicate a generic node in the DP lattice.

3.1.1 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 [12] is applied on all the (remaining) wagons, relative to p_e. LiteSEER is based on the "safety function"

$$f(q_a) = c(p_w, q_a) - (1 + \lambda_g^x)c(p_e, q_a) \tag{3.1}$$

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,

$$\forall\ q_a \in Corners(\mathsf{S}),\ f(q_a) \leq 0 \tag{3.2}$$

In Figure 2.1(b), this corresponds to requiring that the replacement be safe at V_0, V_1, V_2 and V_3.

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 [12]).

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

$$\xi(p_w, p_e) = \frac{\overline{c}(p_e, q_a)}{\overline{c}(p_w, q_a)} \qquad q_a \in Corners(\mathsf{S}) \tag{3.3}$$

That is, we use a CornerAvg heuristic wherein a comparison of the arithmetic mean of the costs at the *corners* of S is used as an indicator of the potential assistance that will be provided throughout 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.

Our choice of the CornerAvg heuristic is movitated by the following observations:

- The arithmetic mean favors sub-plans that perform well in the *top-right region* of the selectivity space since the largest cost magnitudes are seen there.[1] We already know that POSP plans in this region tend to have large endo-optimal regions [11] and are therefore more likely to provide good stability as per the discussion in Section 2.2. Here, we assume that this observation holds true for the sub-plans of near-optimal plans as well.

- The arithmetic mean ensures that the Skyline set of wagons, described next, completely represents all the wagons in the candidate set.

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

[1] Assuming that "plan-cost-monotonicity" (PCM) [11] holds, whereby plan costs rise with increased selectivities, as is usually the case in practice.

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 [6] 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 the Appendix.

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.

A note about the sequencing of the four checks – in principle, Check 1 (Local Cost) could be done after the global checks. However, it is carried out first since it is trivial to evaluate, whereas the remaining three checks are all dependent on FPC computations, which are comparatively expensive, and therefore their invocations should be minimized. Further, Checks 2 to 4 require the engine to be identified before they can proceed, whereas Check 1 can be continually executed, using the lowest cost plan seen thus far in the enumeration process as the (pseudo)-engine.

3.1.2 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. Ideally, we would like to set it to a level that ensures maximum stability without falling prey to superfluous replacements. However, there is a secondary consideration – using a lower value and thereby going ahead with some of the stability-superfluous replacements may help to achieve *anorexic* plan diagrams, which as mentioned before, is a potent objective in query optimizer construction. We discuss this issue in more detail in Chapter 5.

3.2 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 ξ.[2]

Constant Ranking Property. An important property of the above selection procedure, borne out by the definition of ξ, is that it always gives the *same ranking* between a given pair of potential replacement plans *irrespective of the specific query* q_e *in* S *that is currently being optimized*. This is exactly how it should be since the stability of a plan vis-a-vis another plan should be determined by its *global* behavior over the entire space.

Example Replacement. To make the plan replacement procedure concrete, consider the example situation shown in Table 3.1, obtained at the root of the DP lattice for query $\hat{Q}10$ using the NodeExpand algorithm with $\lambda_l, \lambda_g = 20\%, \delta_g = 1$. We present in this table the engine (P_1)

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

and *seventy three* additional wagons (P_2 through P_{74}), ordered on their local costs. The corner costs and benefit indices of these plans are also provided, and in the last column, the check (if any) that resulted in their pruning. As can be seen, each of the checks eliminates some wagons, and finally, only two wagons (P_9, P_{19}) survive all the checks. From among them, the final plan chosen is $\boldsymbol{P_{19}}$ which has the maximum ξ = 1.26, and whose local cost (334801) is within 4% of P_1 (322890).

Plan No	Local Cost	V_0 Cost	V_1 Cost	V_2 Cost	V_3 Cost	ξ	Pruned by
P1	**322890**	202089	224599	846630	1271678	1.00	
P2	322901	202101	224610	846642	1271689	0.99	Benefit
P3	323026	202091	224593	905309	1247883	0.98	Benefit
P4	324203	202089	224604	846636	1952627	0.78	Safety
...	...	...	...	...	...	...	...
P9	329089	208207	230766	356555	1280663	1.22	
P10	329100	208219	230777	356567	1280674	1.22	Skyline
P11	329229	202090	224928	846959	4563459	0.43	Safety
...	...	...	...	...	...	...	...
P19	**334801**	**214078**	**236628**	**362417**	**1204051**	**1.26**	
P20	335428	208208	231095	356884	4572444	0.47	Safety
P21	337838	208218	231097	356886	9354574	0.25	Safety
...	...	...	...	...	...	...	...
P32	390748	202208	500856	1866554	12495404	0.17	Cost
P33	395288	202096	228361	850384	38862955	0.06	Cost
...	...	...	...	...	...	...	...
P73	$> 10^{12}$	$> 10^{8}$	$> 10^{12}$	$> 10^{9}$	$> 10^{13}$	< 0.1	Cost
P74	$> 10^{12}$	$> 10^{8}$	$> 10^{12}$	$> 10^{9}$	$> 10^{13}$	< 0.1	Cost

Table 3.1: Example Replacement at Root Node ($\hat{Q}10$)

Expand $(Node\ x, \lambda_l^x, \lambda_g^x, \delta_g)$

$Node\ x$: A node in the DP-lattice
λ_l^x : Local-optimality threshold for node x (set as per Table 4.1)
λ_g^x : Global-safety threshold for node x (set as per Table 4.1)
δ_g : Global-benefit threshold (set as per Table 4.1)

$x.PlanTrain \leftarrow \phi$
$x.ErrorSensitive \leftarrow FALSE$
if SubTree(x) contains at least one error-sensitive relation **then**
    $x.ErrorSensitive \leftarrow TRUE$
end if
if $x.ErrorSensitive = FALSE$ **then**
    {Standard DP}
    $x.PlanTrain \leftarrow$ Optimizer's cheapest plan for computing x
    Return $x.PlanTrain$
else
    {Expansion Process}
    if $x.level = LEAF$ **then**
        $x.PlanTrain \leftarrow$ All possible access paths for base relation i
    else
        for all pairwise node combinations that generate Node x **do**
            Let A and B be the lower level nodes combining to produce x
            Let $A.PlanTrain$ and $B.PlanTrain$ be the plan-trains of
                A and B, respectively.
            for each p_A in $A.PlanTrain$ **do**
                for each p_B in $B.PlanTrain$ **do**
                    $x.PlanTrain \leftarrow x.PlanTrain \cup$ {Plans formed by
                        joining p_A and p_B in all possible ways}
                end for
            end for
        end for
    end if

    {4-stage Pruning Process}
    Let p_e be the engine of $x.PlanTrain$
    {1. Local Cost Check}
    for each wagon plan $p_w \in x.PlanTrain$ **do**
        if $cost(p_w, q_e) > (1 + \lambda_l^x) cost(p_e, q_e)$ **then**
            $x.PlanTrain \leftarrow x.PlanTrain - \{p_w\}$
        end if
    end for
    {2. Global Safety Check}
    for each wagon plan $p_w \in x.PlanTrain$ **do**
        for each point $q_a \in Corners(S)$ **do**
            if $cost(p_w, q_a) > (1 + \lambda_g^x) cost(p_e, q_a)$ **then**
                $x.PlanTrain \leftarrow x.PlanTrain - \{p_w\}$
                break
            end if
        end for
    end for
    {3. Global Benefit Check}
    for each wagon plan $p_w \in x.PlanTrain$ **do**
        $p_w.\xi \leftarrow \frac{\Sigma_{q_a \in Corners(S)} cost(p_e, q_a)}{\Sigma_{q_a \in Corners(S)} cost(p_w, q_a)}$
        if $x.level = ROOT$ **and** $p_w.\xi \leq \delta_g$ **then**
            $x.PlanTrain \leftarrow x.PlanTrain - \{p_w\}$
        else if $x.level \neq ROOT$ **and** $p_w.\xi \leq 1$ **then**
            $x.PlanTrain \leftarrow x.PlanTrain - \{p_w\}$
        end if
    end for
    {4. Skyline Check}
    $x.PlanTrain \leftarrow$ Cost-Safety-Benefit Skyline ($x.PlanTrain$)

    if $x.level = ROOT$ **then**
        $x.PlanTrain \leftarrow$ Plan with Maximum ξ in $x.PlanTrain$
    end if

    Return $x.PlanTrain$
end if

Figure 3.1: Node Expansion Procedure

Chapter 4

Replacement Algorithms

We can obtain a host of replacement algorithms by making different choices for the λ_l^x and λ_g^x settings in the generic process described above. For example, we could choose to keep them constant throughout the lattice. Alternatively, high values could be used at the leaves, progressively becoming smaller as we move up the tree. Or, we could try out exactly the opposite, with the leaves having low values and more relaxed thresholds going up the tree. In essence, a rich design space opens up when stability considerations are incorporated into classical cost-based optimizers.

We consider here a few representative instances that cover a range of tradeoffs between the number and diversity of the candidate replacement plans, and the computational overheads incurred in generating and processing these candidates.

RootExpand. The RootExpand algorithm is obtained by setting both λ_l^x and λ_g^x to 0 at all leaves and internal nodes,[1] while at the root node, these parameters are set to the user's constraints λ_l, λ_g, respectively. This is a simple variant of the classical DP procedure, wherein DP is used as-is starting from the leaves until the final root node is reached. At this point, the competing (complete) plans that are evaluated at the root node are filtered based on the four-check sequence, and a final plan selection is made from the survivors using the procedure described in Section 3.2.

The functioning of RootExpand is pictorially shown in Figure 4.1(a) for the example query

[1]For $\lambda_l^x = 0$, the value of λ_g^x is actually immaterial since it never comes into play.

$\hat{Q}10$ with $\lambda_l, \lambda_g = 20\%$. In this picture, the value above each node signifies the cost of the optimal sub-plan to compute the relational expression represented by the node – for example, the cheapest method of joining ORDERS (O) and LINEITEM (L) has an estimated cost of 313924. At the root node, the second-cheapest plan, NCOL(2), is chosen in preference to the standard DP choice NCOL(1), due to locally being well within 20% of the lowest cost of 322890, and having a BenefitIndex of $\xi = 1.23$.

SkylineUniversal. The SkylineUniversal algorithm is obtained by setting both λ_l^x and λ_g^x to ∞ at the leaves and internal nodes. It represents the other end of the spectrum to RootExpand in that it propagates, beginning with the leaves, *all* wagons evaluated at a node to the levels above. That is, modulo the Skyline Check, which only eliminates redundant wagons, there is absolutely no other pruning anywhere in the internals of the lattice, resulting in the root node effectively processing the *entire set of complete plans* present in the optimizer's search space for the query.

A pictorial representation of SkylineUniversal is shown in Figure 4.1(b) for the same example scenario. In this picture, unfettered expansion is carried out specifically at the nodes that contain one or more error-sensitive relations in their sub-trees, symbolized by the double boxes. Whereas, the standard DP procedure is used in the remainder of the lattice, and this is the reason for only single plans being forwarded, for example, in the N-C sub-lattice component – both leaves, NATION and CUSTOMER, are not error-sensitive relations. The labels above the error-sensitive nodes indicate the various wagons that have survived the four-check procedure, along with their local costs and benefit indices. For example, CO(2) has a cost of 31243 and ξ = 3.24.

In this example, the number of plans enumerated at the root node NCOL is 1099 and 10 of them successfully pass the four-stage check. The plan finally chosen is NCOL(3) which has a cost of 328820 (about 2% more expensive than the cost-optimal NCOL(1)) and provides the maximum BenefitIndex of $\xi = 1.38$.

NodeExpand. The NodeExpand algorithm strikes the middle ground between the replacement richness of Universal and the computational simplicity of RootExpand, by "opening the sub-plan pipe" to a limited extent. Specifically, the version of NodeExpand that we evaluate here

sets $\lambda_l^x = \lambda_l, \lambda_g^x = \lambda_g$ at all error-sensitive nodes – that is, the root node's cost constraints are inherited at the lower levels as well. These settings are chosen to ensure that the *sub-plans* also provide the same local-optimality and global-safety guarantees as the complete plan, a feature that we expect would prove useful in real-world environments with issues such as run-time resource consumption. Further, as a useful byproduct, these settings also help to keep the expansion overheads under control during optimization.

An example of NodeExpand is shown in Figure 4.1(c), where 3 plans survive the four-stage check at the root, and NCOL(3) whose BenefitIndex of 1.26 is the highest, is chosen as the final selection.

The constraints imposed by the three expansion algorithms presented above are summarized in Table 4.1 – standard DP is also included for comparative purposes.

Optimization Algorithm	**Leaf Node** λ_l^x, λ_g^x	**Internal Node** λ_l^x, λ_g^x	**Root Node** λ_l^x, λ_g^x	δ_g
Standard DP	0	0	0	–
RootExpand	0	0	λ_l,λ_g	≥ 1
NodeExpand	λ_l,λ_g	λ_l,λ_g	λ_l,λ_g	≥ 1
SkylineUniversal	∞	∞	λ_l,λ_g	≥ 1

Table 4.1: Constraints of Plan Replacement Algorithms

4.1 Reducing Expansion Overheads

As discussed above, the EXPAND algorithms permit, in general, a train of wagons to be propagated from each node to the upper levels in the lattice. Due to the multiplicative nature of the DP tree, the computational and resource overheads arising out of these additional wagons, if not carefully regulated, can quickly spiral out of control. We have already discussed how expansion is not carried out at the error-insensitive nodes of the DP lattice. We now describe another optimization that also serves to substantially reduce the overheads.

Inheriting Engine Costs for Wagons. 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*. Therefore, we need to only combine the two engines in all possible ways, just like in standard DP, and then simply reuse these associated costs to evaluate the total costs for all other pairings between the two trains.

Further, the inheritance strategy can be used not just for the local costs, but for the remote FPC-based corner costs as well.

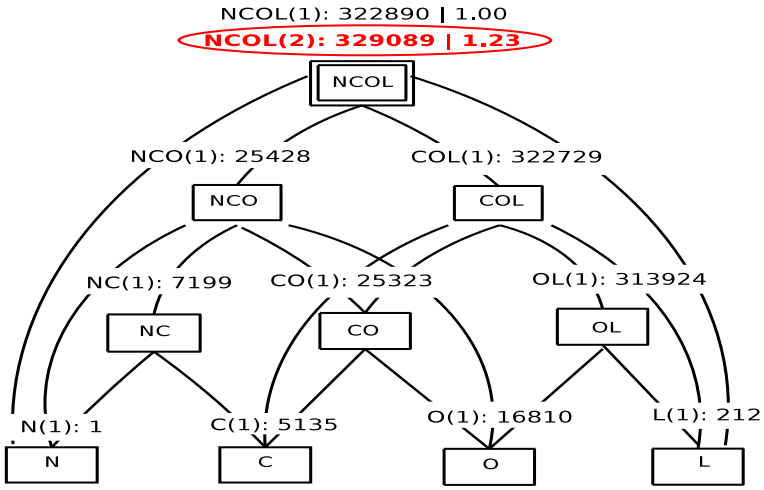
4.2 Comparison with SEER

In our earlier SEER approach [12], we had attempted to identify robust plans through the *anorexic reduction of plan diagrams*. There are some critical differences between this earlier "offline reduction" approach and our current "online production" work:

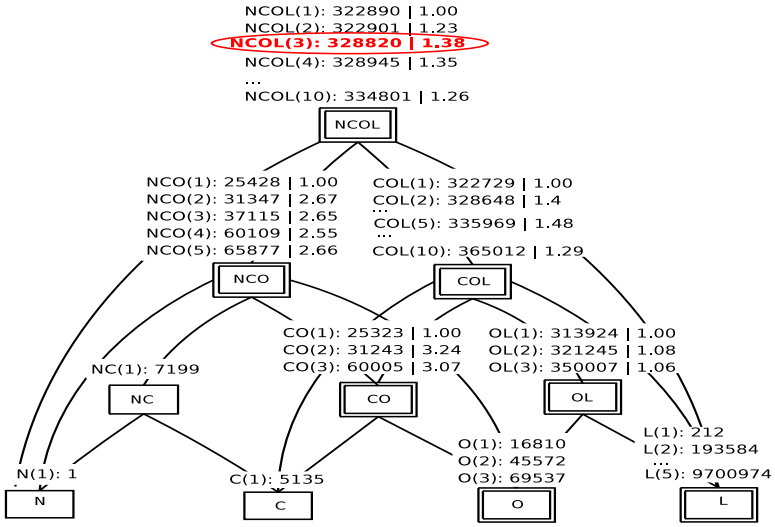
(i) Our techniques are applicable to *ad-hoc individual queries*, whereas the SEER approach is useable only on form-based query templates for which plan diagrams have been previously computed.

(ii) Unlike SEER, our choice of replacement plans is not restricted to be only from the parametric optimal set of plans (POSP). In principle, it could be *any other plan* from the optimizer's search space that satisfies the user's cost constraints. For example, a very good plan that is always second-best by a small margin over the entire selectivity space. In this case, SEER would, by definition, not be able to utilize this plan, whereas it would certainly fall within our ambit. This is confirmed in our experimental study (Chapter 5), where we find that non-POSP plans do regularly feature in the set of recommended plans.

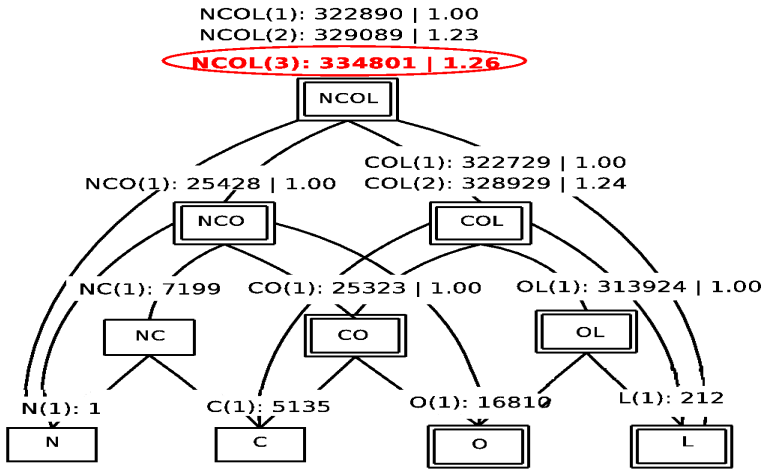
(iii) Finally, as previously mentioned, since SEER considers only POSP plans, it is possible that a chosen replacement may be internally composed of sub-plans that are either not near-optimal or not stable with regard to the corresponding cost-optimal sub-plan. In contrast, an attractive feature of NodeExpand is that it ensures performance fidelity of the replacement throughout its operator tree.



(a) RootExpand ($\lambda_l, \lambda_g = 20\%$)



(b) SkylineUniversal ($\lambda_l, \lambda_g = 20\%$)



(c) NodeExpand ($\lambda_l, \lambda_g = 20\%$)

Figure 4.1: Plan Expansion Algorithms ($\hat{Q}10$)

Chapter 5

Experimental Results

The replacement algorithms described in the previous sections were implemented in PostgreSQL 8.3.6 [23] 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 8.04. In this section, we first outline the experimental framework used to evaluate the performance characteristics of these algorithms, and then highlight the results of the study. The user-specified cost-increase threshold in all our experiments was $\lambda_l, \lambda_g = 20\%$, an acceptable value in practice as per our discussions with industrial development teams, and also a value found sufficient to provide anorexic plan diagrams in popular commercial optimizers [11, 12]. With regard to the benefit threshold δ_g, the default value is the minimum of 1, but we discuss the implications of alternative settings.

Query Templates and Plan Diagrams. 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 $\hat{Q}10$ as parameters. These templates, listed in [1], are all based on queries appearing in the **TPC-H** and **TPC-DS** benchmarks, and cover both 2D and 3D selectivity spaces. They feature a variety of advanced SQL constructs including groupings, orderings, nested queries, correlated predicates, aggregates, functions, 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, while the TPC-DS database hosts skewed data that occupies 100GB.

For each of the query templates, we produced plan diagrams with the Picasso visualiza-

tion tool [28], at a uniformly distributed resolution of 100 in each dimension, resulting in ten thousand queries for 2D templates, and a million queries with 3D templates.

Physical Design. We considered two physical design configurations in our study: **PrimaryKey (PK)** and **AllIndex (AI)**. The PK configuration represents the default physical design of the database engine, wherein a clustered index is created on each primary key. AI, on the other hand, represents an "index-rich" situation with (single-column) indices available on all query-related schema attributes.

Performance Metrics. A variety of performance metrics are used to characterize the behavior of the various replacement algorithms:

1. **Plan Stability:** The overall effect of plan replacements on stability is measured through the AggSERF, MaxSERF and MinSERF 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 zero.

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* [11, 12]. We also tabulate the number of non-POSP plans selected by our techniques.

3. **Computational Overheads:** This metric computes the average overheads incurred, with regard to both time and space, by the various replacement algorithms, relative to those incurred by the standard DP procedure.

Query Template Descriptors. In the subsequent discussion, we use **QTx** to denote a query template based on Query *x* of the TPC-H benchmark, and **DSQTx** to refer to a query template based on Query *x* of the TPC-DS benchmark. 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.

Query Template	RootExpand REP %	RootExpand Agg SERF	RootExpand Max SERF	RootExpand Help %	NodeExpand REP %	NodeExpand Agg SERF	NodeExpand Max SERF	NodeExpand Help %	SkylineUniversal REP %	SkylineUniversal Agg SERF	SkylineUniversal Max SERF	SkylineUniversal Help %	SEER REP %	SEER Agg SERF	SEER Max SERF	SEER Help %
QT5	84	0.54	1	55	85	0.54	1	55	85	0.54	1	55	47	0.61	1	64
QT8	42	0.11	1	1	84	0.13	1	3	–	–	–	–	39	-0.09	1	1
QT10	32	0.20	1	19	98	0.21	1	20	98	0.21	1	20	37	0.21	1	20
AIQT5	87	0.37	1	36	99	0.37	1	38	–	–	–	–	87	0.38	1	39
3DQT8	47	0.17	1	16	69	0.18	1	18	–	–	–	–	59	0.17	1	18
3DQT10	30	0.37	1	67	99	0.39	1	71	99	0.39	1	71	24	0.38	1	41
AI3DQT8	30	0.18	1	21	98	0.19	1	21	–	–	–	–	55	0.12	1	15
AI3DQT10	30	0.11	1	13	99	0.13	1	19	–	–	–	–	55	0.11	1	13
DSQT7	93	0.28	1	28	93	0.28	1	28	93	0.28	1	28	46	0.28	1	28
DSQT18	12	0.31	1	33	58	0.48	1	49	–	–	–	–	57	0.48	1	49
DSQT26	30	0.48	1	50	30	0.49	1	50	30	0.49	1	49	29	0.49	1	49
AIDSQT18	11	0.03	1	3	75	0.07	1	5	–	–	–	–	68	0.04	1	8

Table 5.1: Plan Stability Performance

5.1 Plan Stability Performance

The stability performance results of the RootExpand, NodeExpand, SkylineUniversal and SEER algorithms are enumerated in Table 5.1 for a representative set of a dozen query templates from our study that cover a spectrum of error dimensionalities, benchmark databases, physical designs and query complexities.

Our initial objective here was to evaluate whether there is really tangible scope for plan replacement or whether the optimizer's plan itself is usually the robust choice. We see in the table that the percentage of query points where plan replacement does occur with RootExpand and NodeExpand is quite substantial, even reaching in *excess of 90%* for some templates (e.g. DSQT7)! On average across all the templates, the replacement percentage was around 40% for RootExpand and 80% for NodeExpand.

Not all of the above replacements are required for achieving stability, and the stability-superfluous replacements could be eliminated by setting higher values of δ_g. For example, in AIQT5, $\delta_g = 1.05$ achieves the same stability as the default $\delta_g = 1$ and brings the replacement percentage down from 99% to only 32% for NodeExpand. However, the additional replacements are useful from a different perspective – they help to produce anorexic plan diagrams, as seen later in this section. For example we see that by setting δ_g to a higher value for AIQT5, the number of plans jumps up from 7 to 20 showing the loss of anorexia in plan diagrams.

Moving on to the stability performance, we observe that the AggSERF values of both RootExpand and NodeExpand are usually in the range of 0.1 to 0.6, with the average being about

0.3, which means that on average about *one-third* of the performance handicap due to selectivity errors is removed. Further, a deeper analysis leads to an even more positive view: Firstly, consider the Help% statistics, where we see that, for several templates, a significant fraction of the error population *do receive substantial assistance*. For example, AIQT5 has the performance gap closed by more than two-thirds in about 40 percent of its error situations. A sample frequency distribution of the positive SERF values obtained with 3DQT10, which has the best Help% of over 70%, is shown in Figure 5.1. It is clearly evident in this graph that a sizeable fraction (~20%) receive SERF in excess of 0.9 – i.e., effectively achieve immunity from the errors.

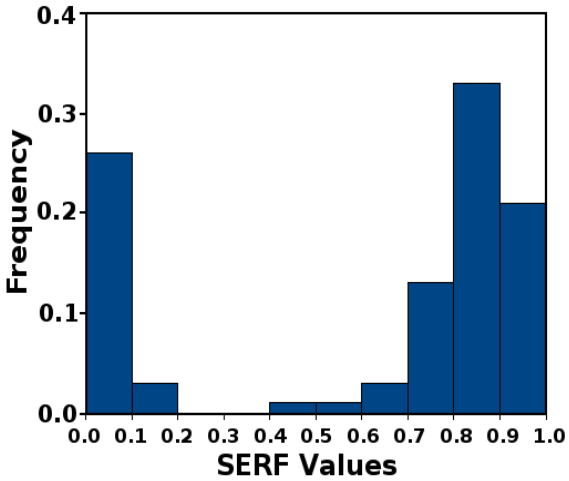


Figure 5.1: Frequency Distribution of SERF values

Second, notice that the AggSERF performance of the offline SEER is largely similar to that of RootExpand and NodeExpand. In our prior study [12], SEER had produced better results for these same templates – the difference is that those experiments were carried out on a popular commercial optimizer, considerably more sophisticated than PostgreSQL. It therefore supported a much richer space of quality replacements. Our expectation is that when our algorithms are implemented in such high-end optimizers, both the AggSERF and Help% values will noticeably increase.

Third, we see that the performance of RootExpand and NodeExpand, in spite of considering a much smaller set of replacement candidates, is virtually identical to that of SkylineUniversal in

the templates where it was able to successfully complete (the templates for which the algorithm ran out of memory are shown with -).

Finally, looking at the MaxSERF metric, we find that in all the templates, there do exist error situations wherein complete immunity (MaxSERF = 1) is obtained from the error, testifying to the inherent power of the replacement approach.

However, on the down side, there certainly are templates such as AIDSQT18, where the AggSERF values are extremely low – however, our investigation suggests that this is more a characteristic of the templates than a flaw in our approach since even the yardstick algorithms, SEER and SkylineUniversal, are unable to achieve useful improvements on these templates.

Taken in toto, the above results suggest that the controlled expansion technique is highly competitive with algorithms that possess complete knowledge of the optimality and search spaces. This leads us to speculate that our approach is successful in extracting most of the benefits, where available, of plan replacement.

5.2 Plan Safety Performance

We now shift our attention to the MinSERF metric to evaluate the *safety* aspect of plan replacement. The results are presented in Table 5.2 and we see that for both RootExpand and NodeExpand: (a) only a few templates have negative values below $-\lambda_g$ (-0.2), (b) even in these cases, the harmful replacements (shown as **Harm%**) occur for only a miniscule percentage of error locations (less than 1% for 2D templates and less than 5% for 3D templates), and (c) most importantly, their magnitudes are small – the lowest MinSERF value is within -5. (The reason that even SEER, which is supposed to guarantee safe replacements, has a few minor negative MinSERF values is that, in order to maximize its scope for replacement, we implemented it also with the LiteSeer heuristic.)

5.3 Plan Diagram Characteristics

We now move on to investigating the nature of the *plan diagrams* obtained with the replacement algorithms. These results are shown in Table 5.3 for the various query templates and replace-

Query Template	RootExpand Min SERF	RootExpand Harm %	NodeExpand Min SERF	NodeExpand Harm %	SkyUniv Min SERF	SkyUniv Harm %	SEER Min SERF	SEER Harm %
QT5	0	0	0	0	0	0	-0.01	0
QT8	0	0	0	0	–	–	0	0
QT10	-0.24	0.25	-0.24	0.01	-0.24	0.51	-0.25	0.20
AIQT5	0	0	0	0	–	–	0	0
3DQT8	-1.05	0.01	-2.30	0.01	–	–	0	0
3DQT10	-1.08	1.93	-0.78	2.15	-0.78	2.15	-0.76	0.01
AI3DQT8	-4.88	0.43	-2.80	4.30	–	–	0	0
AI3DQT10	-2.08	1.74	-4.20	0.54	–	–	-0.69	0.01
DSQT7	0	0	0	0	0	0	0	0
DSQT18	0	0	0	0	–	–	0	0
DSQT26	0	0	0	0	0	0	0	0
AIDSQT18	0	0	0	0	–	–	0	0

Table 5.2: Plan Safety Performance

Query Template	DP Plans	RootExpand Plans	RootExpand Non-POSP	NodeExpand Plans	NodeExpand Non-POSP	SkyUniv Plans	SkyUniv Non-POSP	SEER Plans
QT5	11	3	0	3	0	3	0	2
QT8	18	15	11	3	0	–	–	2
QT10	15	7	1	3	0	3	0	2
AIQT5	29	13	3	7	4	–	–	4
3DQT8	43	22	17	3	0	–	–	2
3DQT10	30	12	2	5	1	5	1	3
AI3DQT8	70	51	41	14	12	–	–	7
AI3DQT10	83	37	5	26	17	–	–	7
DSQT7	12	3	1	2	1	2	1	2
DSQT18	17	23	8	2	1	–	–	2
DSQT26	13	9	7	2	1	2	1	2
AIDSQT18	28	31	7	3	1	–	–	3

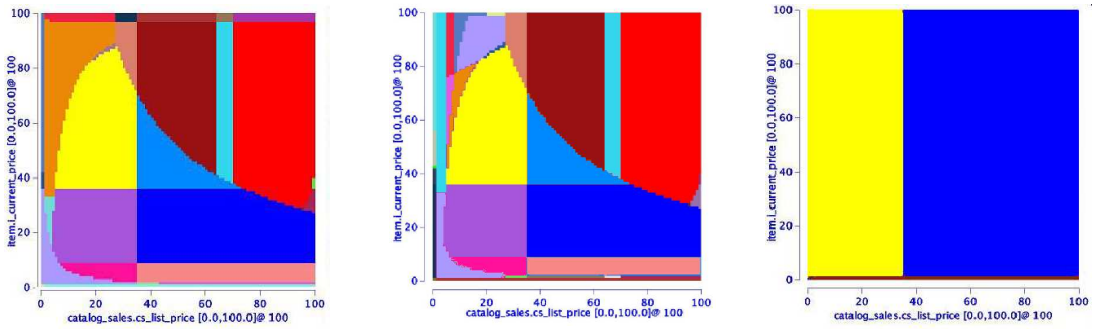
Table 5.3: Plan Diagram Performance

ment algorithms – to put the values in context, we also show the number of plans obtained with the standard DP-based optimizer.

Plan Diagram Cardinality. We see in Table 5.3 that RootExpand generally has rather large plan cardinalities, sometimes eliminating only a few plans (e.g. in QT8, the 18 plans of DP are brought down to 15), or worse, occasionally even *increasing* the number of plans beyond that of DP (e.g. in AIDSQT18, DP has 28 plans whereas RootExpand features 31 plans)!

The NodeExpand algorithm, on the other hand, consistently delivers *anorexic plan diagrams*, with the number of plans in the neighbourhood of ten for virtually all the templates. In fact, its plan cardinality is often comparable to that of SEER – this is quite gratifying since it is obtained in spite of having (a) a much richer space from which to choose replacements, and (b) no prior knowledge of the choices made at other points in the space.

A sample set of plan diagrams produced on the AIDSQT18 template by DP, RootExpand



(a) **DP: 28 plans** (b) **RootExpand: 31 plans** (c) **NodeExpand: 3 plans**

Figure 5.2: **Sample Plan Diagrams for DP, RootExpand and NodeExpand (AIDSQT18,** $\lambda_l, \lambda_g = 20\%, \delta_g = 1$**)**

and NodeExpand are shown in Figures 5.2(a) – 5.2(c). [1] These pictures vividly demonstrate that NodeExpand delivers anorexic diagrams in addition to good plan robustness, whereas RootExpand is only capable of providing the latter.

An isolated exception to NodeExpand's anorexic performance is AI3DQT10 where 26 plans feature, whereas SEER is able to restrict it to 7. However, this can be remedied if λ_l^x and λ_g^x are increased to 100% (from the default 20%) at the internal nodes – that is, if the size of the sub-plan pipe is increased, we again obtain anorexic diagrams with the number of plans coming down to 16. The tradeoffs here are (a) a marginally reduced AggSERF of 0.07, (b) weakened sub-plan performance guarantees, and (c) about 10% increased memory consumption to accomodate the larger pipe.

Yet another observation is of relevance here – the top 10 plans, area-wise, of the above-mentioned 26 plans, collectively cover more than *99%* of the plan diagram. This means that the remaining plans occur in very few locations and if we assume that all queries are equally probable, these small-area plans are unlikely to be encountered in practice, thereby approximating anorexia. With standard DP on the other hand, 70 of the 83 plans are required for a similar area coverage!

Non-POSP plans. We note in Table 5.3 that for each of the expansion algorithms, the fraction

[1] We recommend viewing these diagrams directly from the color PDF file, or from a color print copy, since the greyscale version may not clearly register the various features.

Query Template	Optimization Time (ms)				
	DP	RootExpand		NodeExpand	
QT5	5.4	7.5	(+2.1)	18.9	(+13.5)
QT8	6.0	9.6	(+3.6)	17.8	(+11.8)
QT10	1.5	3.3	(+1.8)	4.8	(+3.3)
AIQT5	6.8	9.7	(+2.9)	20.9	(+14.1)
3DQT8	6.0	20.1	(+14.1)	26.4	(+20.4)
3DQT10	1.5	5.6	(+4.1)	8.1	(+6.6)
AI3DQT8	7.0	27.0	(+20.0)	28.0	(+21.0)
AI3DQT10	1.9	8.2	(+6.3)	10.6	(+8.7)
DSQT7	2.2	3.8	(+1.6)	6.6	(+4.4)
DSQT18	5.0	8.4	(+3.4)	15.7	(+10.7)
DSQT26	2.1	3.5	(+1.4)	6.6	(+4.5)
AIDSQT18	8.6	15.5	(+6.9)	29.8	(+21.2)

Table 5.4: Time Overheads (in milliseconds)

of non-POSP plans in their respective plan diagrams can be quite significant, especially in the heavily-indexed AI environments. As a case in point, with AI3DQT8, there are 41 non-POSP plans out of 51 for RootExpand, occupying 78% of the space, while NodeExpand has 12 on 14, covering more than 90% area.

Usually, the non-POSP fraction is highest for RootExpand and this is attributable to POSP replacements often not being available for consideration at the root node as they have been pruned earlier in the DP lattice (our measurements suggest that this situation occurs in about half the cases).

5.4 Computational Overheads

We now turn our attention to the price to be paid for providing plan stability and anorexic diagrams in terms of increased optimization overheads. The time aspect is captured in Table 5.4 where the per-query optimization times (in milliseconds) are shown for DP, RootExpand and NodeExpand – the additional increase in overheads relative to DP are also shown in parentheses. The results indicate that the performance of both replacement algorithms is within 10 to 20 milliseconds of DP for all the templates. This good performance can be attributed to the techniques incorporated for controlling expansion overheads (Section 4.1).

The numbers in Table 5.5 indicate the additional memory consumption is well within 10MB (for RootExpand) and 100MB (for NodeExpand) over all the query templates. These overheads appear quite acceptable given the richly-provisioned computing environments in vogue today.

Query Template	Memory Overhead (MB)				
	DP	RootExpand		NodeExpand	
QT5	2.0	2.6	(+0.6)	6.2	(+4.2)
QT8	2.0	2.8	(+0.8)	14.8	(+12.8)
QT10	1.6	1.9	(+0.3)	3.9	(+2.3)
AIQT5	2.7	3.5	(+0.8)	11.8	(+9.1)
3DQT8	2.0	5.2	(+3.2)	29.5	(+27.5)
3DQT10	1.6	2.5	(+0.9)	5.0	(+3.4)
AI3DQT8	2.8	6.8	(+4.0)	70.5	(+67.7)
AI3DQT10	1.7	3.6	(+1.9)	7.3	(+5.6)
DSQT7	1.7	2.2	(+0.5)	4.1	(+2.4)
DSQT18	2.0	2.9	(+0.9)	31.0	(+29.0)
DSQT26	1.7	2.1	(+0.4)	4.0	(+2.3)
AIDSQT18	3.2	4.0	(+0.8)	17.0	(+13.8)

Table 5.5: Memory Consumption (in MB)

Further, note that this memory consumption is incurred only for a very brief period of time, much less than one-tenth of a second, as per the statistics in Table 5.4.

Pruning Analysis. As presented in Chapter 4, our expansion algorithms involve a four-stage pruning mechanism, comprising of Cost, Safety, Benefit and Skyline checks. We show in Table 5.6, a sample instance of the collective ability of these checks to reduce the number of wagons forwarded from a node to a limited viable number. In this table, obtained from the root node of a QT8 instance located at (30%, 30%) in S, we show the initial number of candidate wagons, and the number that remain after each check. As can be seen, there are almost 450 plans at the beginning, but this number is pruned to less than 10 by the completion of the last check.

Initial # of Wagons	After			
	Local Cost	Global Safety	Global Benefit	C-S-B Skyline
446	**214**	**194**	**139**	**6**

Table 5.6: Impact of 4-stage Wagon Pruning

Chapter 6

Related Work

Over the last decade, a variety of compile-time strategies have been proposed for identifying robust plans, including the Least Expected Cost [8, 9], Robust Cardinality Estimation [3] and Rio [4, 5] approaches. These techniques provide novel and elegant formulations, but, as described in detail in [12], are limited on some important counts: First, they do not all retain a guaranteed level of local optimality in the absence of errors. That is, at the estimated query location, the substitute plan chosen may be *arbitrarily poor* compared to the optimizer's original cost-optimal choice. Second, these techniques have not been shown to provide sustained acceptable performance *throughout* the selectivity space, i.e., in the presence of arbitrary errors. Third, they require *specialized* information about the workload and/or the system which may not always be easy to obtain or model. Finally, their query capabilities may be *limited* compared to the original optimizer – e.g., only SPJ queries with key-based joins were considered in [3, 4].

Both our previous offline SEER technique, and the online algorithms proposed in this report, address the above limitations through a confluence of (i) mathematical models sourced from industrial-strength optimizers, (ii) combined local and global constraints, and (iii) generic but effective heuristics. The salient differences between SEER and EXPAND were discussed in detail earlier in the report (Section 4.2).

Chapter 7

Conclusions and Future work

We have investigated, in this report, the systematic introduction of global stability criteria in the cost-based DP query optimization process, with a view to minimizing the impact of selectivity errors. Specifically, we proposed the Expand parametrized family of algorithms for striking the desired balance between the competing demands of enriching the candidate space for replacement plans, and the computational/resource overheads involved in this process. The Expand approach is based on expanding the set of plans sent from each node in the DP lattice to the higher levels, subject to a four-stage checking process that ensures only plausible candidates are forwarded, and simultaneously restrict the expansion overheads to an acceptable level.

We considered three plan replacement algorithms: RootExpand, NodeExpand and SkylineUniversal, that cover the spectrum of design tradeoffs. These were implemented in the PostgreSQL kernel, and evaluated through an extensive set of experiments on benchmark environments, which covered a variety of logical and physical designs. Our results showed that a significant degree of robustness can be obtained with relatively minor conceptual changes to current optimizers, especially those that already support a foreign-plan-costing feature. Among the replacement algorithms, NodeExpand proved to be an excellent all-round choice, simultaneously delivering good stability, anorexic plan diagrams, and acceptable computational overheads. The typical situation was that its plan replacements were often able to reduce by more than two-thirds of the adverse impact of selectivity errors for a substantial number of error situations, in return for investing relatively minor additional amounts of optimization time and

memory.

In our future work, we plan to investigate statistical and machine-learning-based techniques for identifying customized assignments to the node-specific cost, safety and benefit thresholds in the Expand approach. Further, while we have assumed that the errors are uniformly distributed over the selectivity space, it would be interesting to consider the more generic case where the error locations have a skewed distribution.

References

[1] 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*, 2009.

[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] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 2005.

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

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

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

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Y. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1991.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] *http://publib.boulder.ibm.com/infocenter/db2luw/ v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/t0*

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

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

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

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

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

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

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

Appendix

In Chapter 3, we described a four-stage pruning procedure that is invoked at each node. The last check in this procedure selectively retains only the *skyline* set of wagons based on cost-safety-benefit considerations. We prove here that the final plan choices made by the optimizer using this restricted set of wagons is exactly equivalent to that obtained by retaining the entire set of wagons – that is, there is no "information loss" due to the pruning.

Theorem 1 *A sub-plan p_w eliminated by the Skyline check cannot feature in the final replacement plan P_{re} selected by the optimizer in the absence of this check.*

Proof: We demonstrate this proof by negation. That is, assume in the absence of the Skyline check, the final plan P_{re} does contain a wagon p_{w1} eliminated by this check. Let the elimination have occurred due to domination by p_{w2} on the dimensionality space comprised of $LocalCost$, $Cost(V_1)$, $Cost(V_2)$, $Cost(V_3)$, $\ldots Cost(V_{2^n} - 1)$, $BenefitIndex$.

Now, let us assess the relationship that develops between p_{w1} and p_{w2} had both been retained through the higher levels of the DP lattice. For example, at the next higher node x, the costs and benefits of the wagons will be

Wagon	Local Cost	Corner Costs	Benefit Index
$w1$	$c(p_{w1}, q_e)$ + Δ_e	$c(p_{w1}, V_i)$ + Δ_{V_i}	$c(p_{w1}, V_i) + \sum \Delta_{V_i}$
$w2$	$c(p_{w2}, q_e)$ + Δ_e	$c(p_{w2}, V_i)$ + Δ_{V_i}	$c(p_{w2}, V_i) + \sum \Delta_{V_i}$

where the deltas are the incremental costs, at the local and corner locations, of computing node x. Note that these incremental costs will be the same for the two wagons since they both represent the same input data and can therefore use the same strategy for computing x.

From the above, it is clear that the relative values along all skyline dimensions have indeed come closer together due to the presence of the additive constants – that is, there is a tighter "coupling". However, there is no "inversion" on any dimension due to which the domination property could be violated. This is because, as is trivially obvious, given two arbitary numbers v_i and v_j with $v_i > v_j$, and a constant a, it is always true that $v_i + a > v_j + a$.

By induction, the above relationship would continue to be true all the way up the lattice to the root node. Now, in the final selection, the MaxBenefit selection heuristic chooses the wagon with the maximum benefit. Therefore, it would still be the case that the plan with p_{w2} would be preferred over the identical plan with p_{w1} instead since the benefit of the former is greater than that of the latter. Hence our original assumption was wrong. ■