Stability-conscious Query Optimization

M. Abhirama Sourjya Bhaumik Atreyee Dey Harsh Shrimal Jayant Haritsa

Technical Report
TR-2009-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India
`http://dsl.serc.iisc.ernet.in`

Oct 5, 2009

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 paper 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.

1 Introduction

In modern database engines, query optimizers choose their execution plans largely based on the classical System R strategy [21]: 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; (iii) finally, choose the plan with the lowest estimated cost.

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 [23], 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 [1], feedback-based adjustments [23, 8], and on-the-fly reoptimization of queries [17, 19, 3]. The particular approach we explore here is to identify, at optimization-time, *robust plans* whose costs 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* in the presence of errors located 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 paper 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 paper, 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.

In essence, our objective is to design a multi-metric (cost and stability) query optimizer. Multi-metric considerations in optimizers are not an entirely new concept – for example, PostgreSQL itself supports using a combination of response time and latency to select execution plans. However, a critical and fundamental difference in our work is the following: Our second metric, stability, is a *global* criterion whereas previous multi-metrics have all been *local*, relevant only to the specific query instance under consideration.

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 [13], 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.

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* [5]. 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** [13], 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 PostgreSQL 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 intra-optimizer online algorithms turns out to be competitive to that of (the extra-optimizer/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* [12] 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 [12] – for example, it enhances the feasibility of parametric query optimization (PQO) techniques [14, 15].

Creating anorexic plan diagrams is a relatively simple matter when the original plan diagram is apriori available. However, it is a much harder task in our environment since we operate within the scope of *individual queries* – this means that the plan choice at a given location is decided with absolutely no knowledge of the choices that would be made at other locations in the selectivity space.

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.)

We also hasten to add that 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 [16], 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 [11] for addressing selectivity errors in the higher nodes of the plan tree.

Contributions. In summary, we present a framework in this paper 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 and intra-optimizer identification of stable query execution plans that provide both guaranteed local near-optimality and enhanced global-stability in an efficient manner on industrial-strength environments.

Organization. The remainder of this paper is organized as follows: In Section 2, we describe the overall problem framework and motivation. The basic EXPAND approach is outlined in Section 3, and representative plan selection algorithms based on this approach are presented in Section 4. Algorithmic extensions to handle various query complexities are discussed in Section 5, while the details of the implementation in PostgreSQL are provided in Section 6. The experimental framework and performance results are highlighted in Section 7. Related work is reviewed in Section 8. Finally, in Section 9, we summarize our conclusions and outline future research avenues.

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 paper.

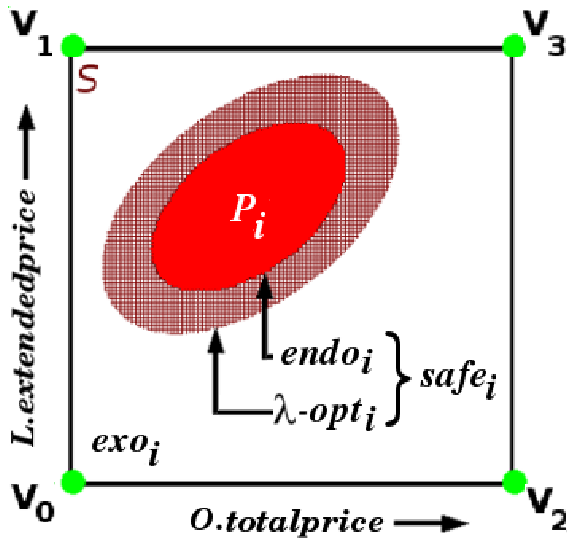
Consider the situation where the user has submitted a query and desires stability with regard to selectivity errors on one or more of the base relations that feature in the query. The choice of the relations could be based on user preferences and/or the optimizer's expectation of relations on which selectivity 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 1(a), an SPJ version of Query 10 from the TPC-H benchmark – this query has four base relations (NATION (N), CUSTOMER (C), ORDERS (O), LINEITEM (L)), two of which – ORDERS, LINEITEM – are deemed to be error-sensitive relations. For this query, the associated 2D error selectivity space S is shown in Figure 1(b).

The d-dimensional selectivity space is represented by a finite dense grid of points wherein each point $q(x_1, x_2, \ldots, x_d)$ corresponds to a query instance with selectivity x_j in the j-th dimension. 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 as V_k, with k being the binary representation of the location coordinates – e.g. the bottom-right corner $(1, 0)$, in Figure 1(b) 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 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 (for reasons explained below) as the plan's *SafeRegion*, denoted by $safe_i$.

```
select  C.custkey, C.name, C.acctbal, N.name, C.address, C.phone
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 1: **Example Query and Selectivity Space**

2.1 Cost Constraints on Plan Replacement

Consider a specific query instance whose optimizer-estimated location in S is q_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 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 $(\lambda_l, \lambda_g > 0)$. Specifically, the user is willing to permit replacement of P_{oe} with P_{re}, iff:

Local Constraint: At the estimated query location q_e,

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

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,

$$\forall q_a \in \mathsf{S} \text{ such that } q_a \neq q_e, \quad \frac{c(P_{re}, q_a)}{c(P_{oe}, q_a)} \leq (1 + \lambda_g) \tag{2}$$

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*.

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 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. (Note that 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 paper.

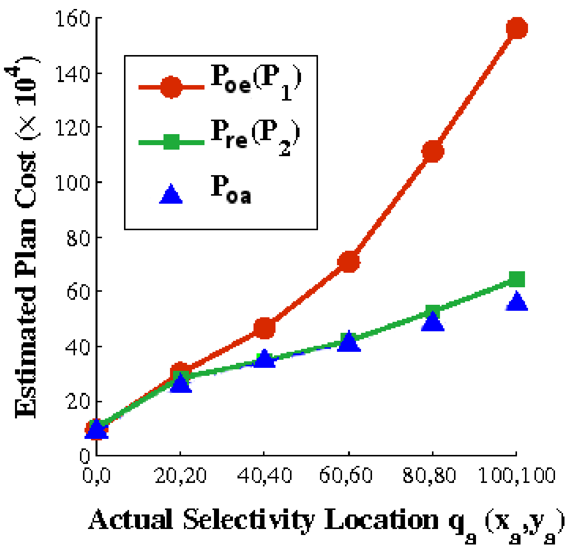
Exo-optimal region of P_{re}: Here, q_a is located outside $safe_{re}$, and at such locations, we cannot apriori predict P_{re}'s behavior relative to P_{oe}– it could range from being much better, substantially reducing the adverse impact of the selectivity error, to the other extreme of being *much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

2.3 Motivational Scenario

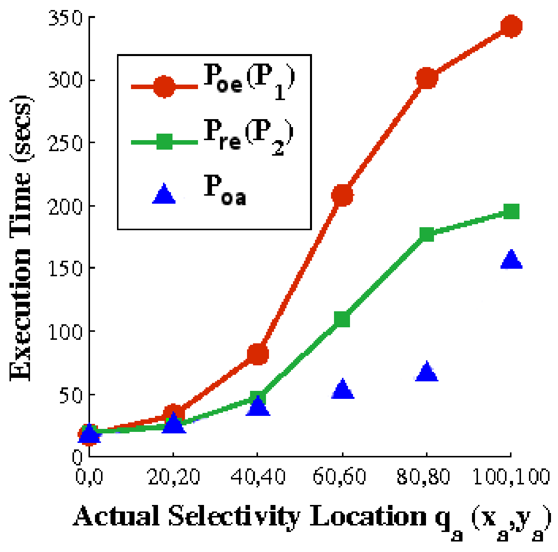
We now present a sample scenario to motivate how plan replacement could help to improve robustness to selectivity errors. Here, the example query $\hat{Q}10$ is input to the PostgreSQL optimizer; the optimizer estimates the query location q_e in S to be $(1\%, 40\%)$, and its cost-optimal choice at this location is plan P_1; and the suggested replacement (by our NodeExpand algorithm with λ_l, λ_g = 20%) is plan P_2. When the costs of these plans are evaluated at a set of error locations q_a – for instance, along the principal diagonal of S, we obtain the graph shown in Figure 2(a). The results indicate that P_2 provides very substantial performance improvements with respect to 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(a)) at each of these locations.

To explicitly assess the compile-time predictions of performance improvements, we *executed* the P_1, P_2 and P_{oa} plans at these various locations – the corresponding response-time graph is shown in Figure 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 – this is an orthogonal research issue that has to be tackled separately.

Incidentally, the difference between P_1 and P_2 is in their join order – the former implements $(C \bowtie (L \bowtie O)) \bowtie N$ while the latter opts for the bushy join $(L \bowtie O) \bowtie (C \bowtie N)$.



(a) **Compile-Time**



(b) **Run-Time**

Figure 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 replacements is based on the **SERF** error resistance metric introduced in [13]. For a specific error instance, corresponding to estimated 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

$$SERF(q_e, q_a) = 1 - \frac{c(P_{re}, q_a) - c(P_{oa}, q_a)}{c(P_{oe}, q_a) - c(P_{oa}, q_a)} \tag{3}$$

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]

$$AggSERF = \frac{\sum_{q_e \in rep(\mathsf{S})} \sum_{q_a \in exo_{oe}(\mathsf{S})} SERF(q_e, q_a)}{\sum_{q_e \in \mathsf{S}} \sum_{q_a \in exo_{oe}(\mathsf{S})} 1} \tag{4}$$

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.

Note that in the above formulation, we assume for simplicity that the actual location q_a is equally likely to be anywhere in P_{oe}'s exo-optimal space, that is, that the errors are randomly distributed over this space. In our future work, we plan to investigate the more generic case where the error locations have an associated probability distribution.

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 it is, by definition, not possible to provide meaningful assistance in the safe region of the optimizer's plan choice P_{oe}, that is, in $safe_{oe}$. However, we still need to consider the possibility that replacements may end up causing *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. $\dfrac{c(P_{re}, q_e)}{c(P_{oe}, q_e)} \leq (1 + \lambda_l)$

[1]In [13], 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.

2. $\forall q_a \in \mathsf{S}$ s.t. $q_a \neq q_e$, $\quad \dfrac{c(P_{re}, q_a)}{c(P_{oe}, q_a)} \leq (1 + \lambda_g)$

   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.

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, we will assume that there are no "interesting order" plans [21] 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. The algorithmic extensions for handling these scenarios are described in Section 5, and are included in our experimental study (Section 7).

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 detailed pseudocode listing is given in Figure 3. 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 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 local cost is more than $(1 + \lambda_l^x)$ times that 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, we evaluate the behaviour of the "safety function", defined as

$$f(q_a) = c(p_w, q_a) - (1 + \lambda_g^x) c(p_e, q_a) \tag{5}$$

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 2)

λ_g^x : Global-safety threshold for node x (set as per Table 2)

δ_g : Global-benefit threshold (set as per Table 2)

```
1:  $x.PlanTrain \leftarrow \phi$
2:  $x.ErrorSensitive \leftarrow FALSE$
3:  if SubTree(x) contains at least one error-sensitive relation then
4:      $x.ErrorSensitive \leftarrow TRUE$
5:  if $x.ErrorSensitive = FALSE$ then
6:      /* Standard DP */
7:      $x.PlanTrain \leftarrow$ {Cheapest plan to compute $x$ + cheapest plan to compute $x$ for each interesting order}
8:      Return $x.PlanTrain$
9:  else
10:     /* Expansion Process */
11:     if $x.level = LEAF$ then
12:         $x.PlanTrain \leftarrow$ All possible access paths for base relation $i$
13:     else
14:         for all pairwise node combinations that generate Node $x$ do
15:             Let $A$ and $B$ be the lower level nodes combining to produce $x$
16:             Let $A.PlanTrain$ and $B.PlanTrain$ be the plan-trains of $A$ and $B$, respectively.
17:             for each $p_A$ in $A.PlanTrain$ do
18:                 for each $p_B$ in $B.PlanTrain$ do
19:                     $x.PlanTrain \leftarrow x.PlanTrain \cup$ {Plans formed by joining $p_A$ and $p_B$ in all possible ways}
20:
21:     for each plan $p$ with interesting order $r$ in $x.PlanTrain$ do
22:         Move $p$ to sub-train $x.PlanTrain_r$.
23:     Move all remaining plans to sub-train $x.PlanTrain_{NO\_ORDER}$.
24:
25:     /* Stem handling for RootExpand */
26:     if (RootExpand) and (isJoinRoot($x$) or isInternalStem($x$)) then
27:         $\lambda_l^x \leftarrow \infty;\ \lambda_g^x \leftarrow \infty$
28:
29:     for each $x.PlanTrain_r$ of node $x$ do
30:         /* 4-stage Pruning Process */
31:         Let $p_e$ be the engine of $x.PlanTrain_r$
32:         /* 1. Local Cost Check */
33:         for each wagon plan $p_w \in x.PlanTrain_r$ do
34:             if $cost(p_w, q_e) > (1 + \lambda_l^x)cost(p_e, q_e)$ then
35:                 $x.PlanTrain_r \leftarrow x.PlanTrain_r - \{p_w\}$
36:         /* 2. Global Safety Check */
37:         for each wagon plan $p_w \in x.PlanTrain_r$ do
38:             for each point $q_a \in Corners(S)$ do
39:                 if $cost(p_w, q_a) > (1 + \lambda_g^x)cost(p_e, q_a)$ then
40:                     $x.PlanTrain_r \leftarrow x.PlanTrain_r - \{p_w\}$
41:                     break
42:         /* 3. Global Benefit Check */
43:         for each wagon plan $p_w \in x.PlanTrain_r$ do
44:             $p_w.\xi \leftarrow \frac{\Sigma_{q_a \in Corners(S)} cost(p_e, q_a)}{\Sigma_{q_a \in Corners(S)} cost(p_w, q_a)}$
45:             if $x.level = ROOT$ and $p_w.\xi \leq \delta_g$ then
46:                 $x.PlanTrain_r \leftarrow x.PlanTrain_r - \{p_w\}$
47:             else if $x.level \neq ROOT$ and $p_w.\xi \leq 1$ then
48:                 $x.PlanTrain_r \leftarrow x.PlanTrain_r - \{p_w\}$
49:         /* 4. Skyline Check */
50:         $x.PlanTrain_r \leftarrow$ C-S-B Skyline ($x.PlanTrain_r$)
51:
52:     if $x.level = ROOT$ then
53:         $x.PlanTrain \leftarrow$ Plan with Maximum $\xi$ in $x.PlanTrain$
54:     Return $x.PlanTrain$
```

Figure 3: Node Expansion Procedure

This function 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, if the cheapest sub-plan were to be (eventually) replaced by the candidate sub-plan, the adverse impact (if any) of this replacement is bounded by λ_g^x – that is, in this sense, it is *safe*. Here, λ_g^x is again 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 $\lambda_g^x \geq \lambda_l^x$.

Evaluating the safety function requires the ability to cost query plans at *arbitrary* locations in the selectivity space. This feature, called "Foreign Plan Costing" (FPC) in [13], is available in commercial optimizers such as DB2 (Optimization Profile), SQL Server (XML Plan) and Sybase (Abstract Plan). For PostgreSQL, we had to implement it ourselves (details in Section 6).

The safety check can be verified by exhaustively invoking the FPC function at *all* locations in S, but the overheads become unviably large. We have recently developed the **CornerCube-SEER (CC-SEER)** [22] algorithm to address this problem. CC-SEER guarantees global safety by merely evaluating the safety function at the *unit hyper-cubes* located at the *corners* of the selectivity space. That is, given a d-dimensional space, FPC costing is carried out at only 4^d points. The intuition here is that, given the nature of plan cost behavior in modern optimizers, if a replacement is known to be safe at the corner regions of the selectivity space, then it is also safe *throughout the interior region* (see [22, 13] for the formal details).

Finally, we have also found that an extremely simple heuristic, called LiteSEER [13], which simply evaluates whether all the *corner points* are safe, that is,

$$\forall\ q_a \in Corners(\mathsf{S}),\ f(q_a) \leq 0 \tag{6}$$

works almost as well as CC-SEER in practice, although not providing formal safety guarantees. In Figure 1(b), this corresponds to requiring that the replacement be safe at V_0, V_1, V_2 and V_3, and in general, requires FPC evaluation only at 2^d points. The experimental study in Section 7 employs a LiteSEER implementation by default, but we also provide sample results with CC-SEER in Section 7.6.

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

$$\xi(p_w, p_e) = \frac{\overline{c}(p_e, q_a)}{\overline{c}(p_w, q_a)} \qquad q_a \in Corners(\mathsf{S}) \tag{7}$$

That is, we use a *CornerAvg* heuristic wherein the arithmetic mean of the costs at the *corners* of S is used as an indicator of the 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 motivated by the following observation: The arithmetic mean favors sub-plans that perform well in the *top-right region* of the selectivity space since the largest cost magnitudes are usually seen there. We already know that POSP plans in this region tend to have large endo-optimal space coverage [12]. Therefore, they are more likely to provide good stability since, *by definition,* any P_{re} provides stability in its own endo-optimal region, as its cost has to be less than that of P_{oe} in this subspace (as discussed previously in Section 2.2). The CornerAvg heuristic projects that this observation holds true for the sub-plans of near-optimal plans as well.

4. Cost-Safety-Benefit Skyline Check: After the above three checks, it is possible that some wagons 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,* right until the lattice root, since the same constants are added to both wagons.

Given the above, it is sufficient to simply use a *skyline* set [5] 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 skyline-based wagon selection technique is equivalent to having retained the entire set of wagons is given in Appendix A.

After 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.

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 the minimum benefit for which replacement is considered a worthwhile option, is a design issue. Ideally, it should be set to ensure 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, a potent objective in query optimizer construction. This issue is discussed in the experimental study of Section 7.

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

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

should be since the stability of a plan vis-a-vis another plan should be determined by its *global* behavior over the entire space.

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

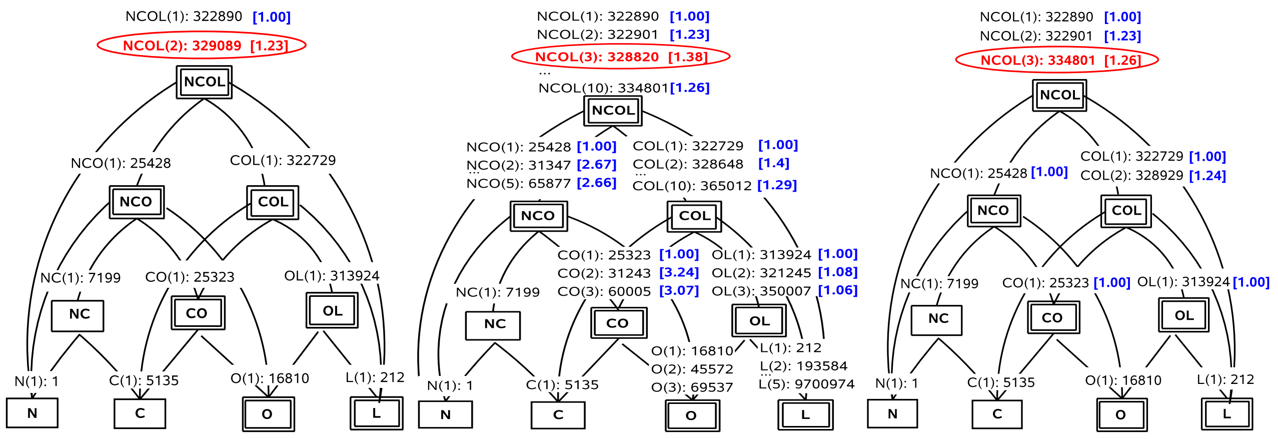
Example Replacement. To make the plan replacement procedure concrete, consider the example situation shown in Table 1, obtained at the root of the DP lattice for query $\hat{Q}10$ using the NodeExpand algorithm with $\lambda_l, \lambda_g = 20\%, \delta_g = 1$. We present in this table the engine (P_1)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

4 Replacement Algorithms

Given the generic process described above, we can obtain a host of replacement algorithms by making different choices for the λ_l^x and λ_g^x settings in the lattice interior. 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, 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



(a) RootExpand (b) SkylineUniversal (c) NodeExpand

Figure 4: Plan Expansion Algorithms ($\hat{Q}10$: $\lambda_l, \lambda_g = 20\%, \delta_g = 1$)

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(a) for the example query $\hat{Q}10$ with $\lambda_l, \lambda_g = 20\%$ (and $\delta_g = 1$). In this picture,the nodes that contain one or more error-sensitive relations in their sub-trees are symbolized by double boxes. Further, 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. Finally, the number in brackets adjacent to each cost at the root node represents the BenefitIndex of the associated plan.

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 the maximum 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. This implies that the root node effectively processes 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(b) for the same example scenario. In this picture, unfettered expansion is carried out at all the error-sensitive nodes (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 plans 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 we expect would prove useful in real-world environments with aspects such as run-time resource consumption. Further, as a useful byproduct, the settings also help to keep the expansion overheads under control.

An example of NodeExpand is shown in Figure 4(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 2 – 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 2: 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. In addition, a crucial optimization for reducing overheads is the following:

Inheriting Engine Costs for Wagons. When two plan-trains arrive and are combined at a node, the cost of combining the engines of the two trains with 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, this cost reuse 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

Our earlier SEER approach [13] identified robust plans through the *anorexic reduction of plan diagrams*. There are fundamental differences between that "offline/extra-optimizer/reduction" approach and our current "online/intra-optimizer/production" work:

(i) Our techniques are applicable to *ad-hoc individual queries*, whereas SEER 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 (Section 7), where non-POSP plans regularly feature in the set of recommended plans.
(iii) Finally, as previously mentioned, an attractive feature of NodeExpand is that it ensures performance fidelity of the replacement throughout its operator tree.

5 Handling Query Complexities

For ease of presentation, we had assumed earlier in Section 3 that optimizing the user query did not feature either (a) "interesting orders" (where a sub-plan produces results in a particular order that could prove useful later in the optimization); or (b) "stems" (where a linear chain of nodes appear above the join root node of the DP lattice). We now discuss the algorithmic extensions necessary to handle these features.

5.1 Interesting Orders

Plans corresponding to interesting orders can be handled by having each train to be composed of not just a single generic sequence of wagons, but instead a *parallel array* of sub-trains, one sub-train for each interesting order. For the sake of uniformity, we treat the set of wagons corresponding to unordered plans to also be part of a generic result order called NO_ORDER.

As discussed earlier, there are two steps to the expansion process – an exhaustive plan enumeration step followed by the four-stage plan retention process. We discuss the changes required in each of the two steps to be able to handle interesting orders.

Plan Enumeration. Let *A* and *B* be a pair of lower level nodes combining together to produce Node x. Then, the plan expansion procedure at Node *x* involves exhaustively combining *all* sub-trains of *A* with *all* sub-trains of *B*. Subsequently, the result order (if any) of each of the newly produced combinations is determined. Combinations with interesting orders are assigned to associated sub-trains, while the unordered combinations are all placed in the NO_ORDER sub-train.

Plan Retention. The plan retention process is handled *independently* for each of the sub-trains and exactly follows the 4-stage pruning procedure described for single trains in Section 3.

5.2 Stems

A stem in a DP-lattice is the linear chain of nodes that may appear above the "join root" node (the node corresponding to the join of all the relations present in the query). The stem usually features aggregation and grouping operators. A sample, based on the example query of Figure 1(a), is shown in Figure 5, where the join root is **NCOL**, and the stem is displayed in the shaded box. The handling of stems is algorithm-specific, as described below.

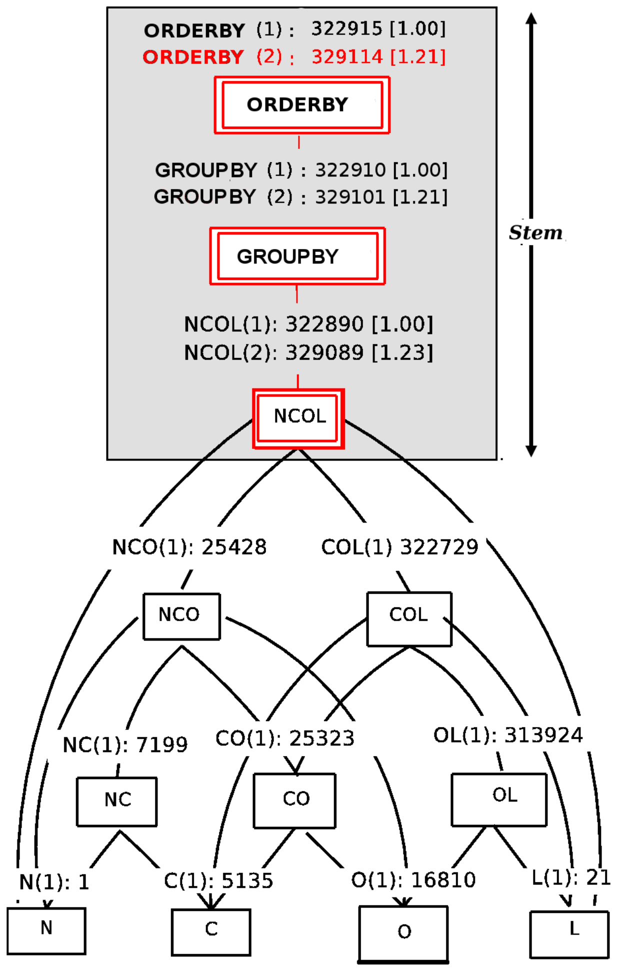


Figure 5: Plan Stem

RootExpand. Here, as explained previously in Section 4, plan expansion takes place only at the terminal node of the DP-lattice, with the rest of the processing in the lattice being identical to standard DP. This is appropriate when the terminal node of the DP-lattice is the join root and there are a set of alternative plans, corresponding to different join orders, to choose from. However, it becomes meaningless if the terminal node is at the end of a stem since only a *single* plan will have survived at this stage in the normal DP process, and therefore the replacement space is virtually non-existent.

We therefore modify the RootExpand algorithm to permit *all* plans that reach the join root to continue to be considered all the way until the terminal node of the stem. That is, λ_l^x and λ_g^x are set to ∞ at the join root and all internal stem nodes that lie between the join root and the terminal node. This procedure is implemented in Lines 26 and 27 of Figure 3.

NodeExpand and SkylineUniversal. For these algorithms, we do not need to make any special changes for handling stems since they, unlike RootExpand, carry out plan expansion at all levels of the DP-lattice, and therefore the stem nodes can be treated in the same way as the canonical lattice nodes.

6 Implementation in PostgreSQL

We have implemented the various algorithms described in the previous sections inside the PostgreSQL [25] kernel, specifically version 8.3.6 [26]. We briefly discuss here the issues related to our implementation experience.

Foreign Plan Costing. In order to implement the LiteSEER and ξ heuristics described in Section 3.2, we need to be able to cost a sub-plan (or plan) at all corners of S. While this feature is present in several commercial optimizers, as mentioned before, it is currently not available in PostgreSQL.

Therefore, we have ourselves implemented remote costing in the PostgreSQL optimizer kernel. Our initial idea was to merely carry out a bottom-up traversal of the operator tree at the foreign location and at each node appropriately invoke the optimizer's costing and output estimation routines. This approach is reasonably straightforward to implement, and more importantly, very efficient.

However, this approach failed to work because PostgreSQL caches certain temporary results during the optimization process which have an impact on the final plan costs – these cached values are not available to a purely offline costing approach. Therefore, we had to monitor and retain sufficient additional information during the current plan generation process such that the cached values for remote locations could be explicitly calculated.

Optimization Process. The PostgreSQL optimizer usually optimizes for a combination of latency and response-time, especially if the access to the output data is through a cursor, or a limit on the number of output tuples is specified. In order to focus our study, we modified the optimization objective to be solely response-time.

Intrusiveness on Code-base. From an industrial perspective, an obvious question is the extent to which the underlying code-base has to be modified to support the proposed approach. In our PostgreSQL implementation, where we have added around 10K lines of code, the vast majority of the additions have gone towards including the FPC feature, which as mentioned before, is already available in most commercial optimizers. Therefore, while we are aware that these systems are considerably more sophisticated than PostgreSQL, our expectation is that incorporating our techniques would be minimally intrusive on their code-base. This is especially true for the RootExpand algorithm, where the behavior of only the final node in the DP lattice is modified.

7 Experimental Results

The replacement algorithms described in the previous sections were implemented in PostgreSQL 8.3.6 [26] 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 thresholds in all our experiments are $\lambda_l, \lambda_g = 20\%$, 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 [12, 13]. 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, enumerated in Appendix B, are all based on queries appearing in the **TPC-H** and **TPC-DS** benchmarks, and cover both 2D and 3D selectivity spaces. The templates 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 (at a uniform grid resolution of 100 on each dimension) with the Picasso visualization tool [31].

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.

Query Template Descriptors. In the subsequent discussion, we use **QTx** and **DSQTx** to label query templates based on Query *x* of the TPC-H benchmark and the TPC-DS benchmark, respectively. 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.

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

1. **Plan Stability and Safety** The effect of plan replacements on stability is measured with the AggSERF 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 wherein opting for a replacement plan reduced the performance gap substantially, specifically, by atleast **two-thirds**.

   Replacement safety is evaluated through MinSERF and the percentage of query locations with MinSERF $< -\lambda_g$ is tabulated.

2. **Plan Diagram Cardinality:** This metric tallies the number of unique plans present in the plan diagram, with cardinalities less than or around *ten* indicating *anorexic diagrams* [12, 13]. 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, relative to those incurred by the standard DP procedure.

7.1 Plan Stability Performance

The stability performance results of the RootExpand, NodeExpand, SkylineUniversal and SEER algorithms are enumerated in Table 3 for a representative set of query templates from our study, which covered a spectrum of error dimensionalities, benchmark databases, physical designs and query complexities.

Our initial objective 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 Table 3 that REP% for both RootExpand and NodeExpand is quite substantial, even reaching in *excess of 90%* for some templates

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	50	29	0.49	1	49
AIDSQT18	11	0.03	1	3	75	0.07	1	5	–	–	–	–	68	0.04	1	8

Table 3: Plan Stability Performance

(e.g. DSQT7)! On average across all the templates, the replacement percentage was around 40% for RootExpand and 80% for NodeExpand.

We hasten to add that not all of these replacements are required for achieving stability, and the stability-superfluous replacements could be eliminated by setting higher values of δ_g. For example, with QT5, $\delta_g = 1.03$ achieves the same stability as the default $\delta_g = 1$ and brings REP% of NodeExpand down from 85% to 32%. Our analysis has shown that in general, about 30%-50% replacements are sufficient to maximize the stability. However, the additional replacements are useful from a different perspective – they help to produce anorexic plan diagrams, as seen later in this section.

Moving on to the stability performance itself, 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. A deeper analysis leads to an even more positive view: Firstly, the Help% statistics indicate that, for several templates, a significant fraction of the error population corresponding to query locations with replaced plans *do receive substantial assistance*. For example, QT5 has the performance gap more than halved in about 55 percent of such error situations. Further, with 3DQT10, which has the best Help% of over 70%, a sizeable fraction (~20%) receive SERF in excess of 0.9 – i.e., effectively achieve *immunity* from the errors.

A sample frequency distribution of the positive SERF values obtained with NodeExpand on 3DQT10, which has the best Help% of over 70%, is shown in Figure 6. It is evident here that a sizeable fraction (~20%) receive SERF in excess of 0.9 – i.e., effectively achieve *immunity* from the errors.

Second, the AggSERF performance of (offline) SEER is quite similar to that of RootExpand and NodeExpand. In our prior study [13], SEER had produced better results for these same templates – the difference is that those experiments were carried out on a sophisticated commercial optimizer supporting a richer space of quality replacements than PostgreSQL. Implementing our algorithms in such high-end optimizers is likely to also significantly increase their AggSERF and Help% contributions.

Third, 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 –). In fact, as shown in Section 7.5, their performance is fairly close to even an *optimal* (wrt AggSERF) version of SkylineUniversal!

Finally, MaxSERF was 1 for all the templates, testifying to the inherent power of the replacement approach.

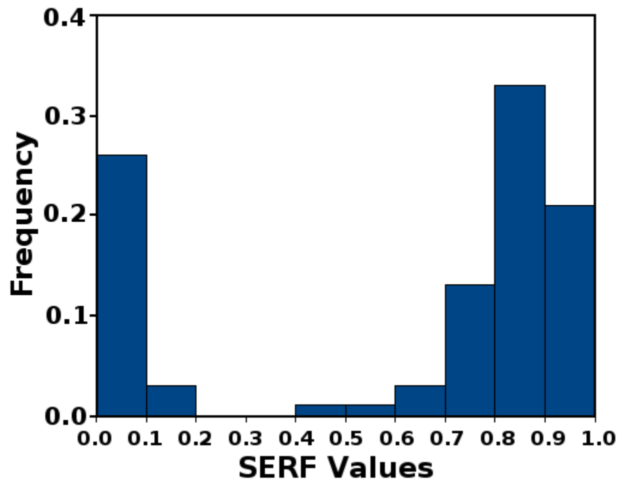


Figure 6: Frequency Distribution of SERF values (3DQT10)

Taken in toto, these results suggest that the controlled expansion technique is capable of extracting most of the benefits obtainable through plan replacement.

Replacement Plan Analysis. We have also conducted a preliminary analysis of the characteristics of the replacement plans vis-a-vis the original choices. Our observations include the following:

- Index intersections are often replaced by joins based on sequential scans. This is due to the indexes becoming very expensive at the higher selectivity regions of the selectivity space. In Table 4, we quantify this change for sample query templates by aggregating the number of index intersections at the replacement locations and comparing to the corresponding number with DP.

Query Template	DP	NodeExpand
AIQT5	493	3
AIQT7	991	691

Table 4: Index Intersection Statistics

- Nested-loop-based plans are frequently replaced with hash-join-based plans, but the reverse was never observed. Further, merge joins were almost never retained. These observations are quantified in Table 5, where we tabulate the aggregate number of times each of the join strategies appears in the replacement plans produced by the NodeExpand approach. The frequency corresponding to standard DP at those replaced points is also shown in Table 5. As can be seen from the results, the hash-join proportion, which was already predominant, tends to eat further into the nested-loop and merge-join presence.

- Finally, we also often saw that the join order of the replacement plan was different to that of the original plan. In particular, left-deep plans were typically replaced by bushy plans.

Query Template	Optimization Algorithm	# Nested Loop Join	# Merge Join	# Hash Join
QT5	DP	288	60	42022
	NodeExpand	242	0	42128
QT10	DP	30	1297	28070
	NodeExpand	30	0	29367
DSQT7	DP	1449	65	35538
	NodeExpand	100	0	36952
DSQT26	DP	2377	2	9557
	NodeExpand	100	0	11836

Table 5: Join Method Usage Statistics

7.2 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 6 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.)

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 6: Plan Safety Performance

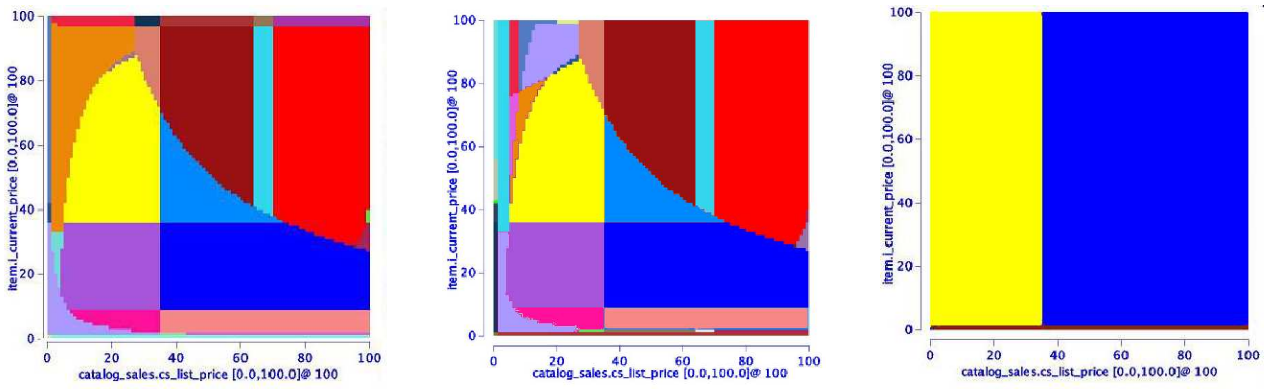
7.3 Plan Diagram Characteristics

We now turn our attention to the characteristics of the *plan diagrams* obtained with the replacement algorithms. The associated results are also shown in Table 3, and to place them in context, the statistics for the standard DP-based optimizer are included.

Plan Diagram Cardinality. We see in Table 7 that for templates such as 3DQT8, where DP generates “dense” diagrams with high plan cardinalities, RootExpand diagrams may also feature a large

Query Template	DP Plans	RootExpand		NodeExpand		SkyUniv		SEER Plans
		Plans	Non-POSP	Plans	Non-POSP	Plans	Non-POSP	
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 7: Plan Diagram Performance



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

Figure 7: **Plan Diagrams for DP, RootExpand, NodeExpand (AIDSQT18, $\lambda_l, \lambda_g = 20\%, \delta_g = 1$)**

number of plans. This behavior is more prevalent in index-rich environments, with the diagram cardinalities even *exceeding* that of DP for some templates – e.g. DP has 28 plans for AIDSQT18, whereas RootExpand features 31 plans!

NodeExpand, on the other hand, consistently delivers strongly *anorexic* plan diagrams for almost all the templates. In fact, its plan cardinality is often comparable to that of SEER – this is quite encouraging since it is obtained in spite of having to contend with (a) a much richer search space from which to choose replacements, and (b) no prior knowledge of the choices made in the remaining selectivity space.

A sample set of plan diagrams produced on the AIDSQT18 template by DP, RootExpand and NodeExpand are shown in Figures 7(a) – 7(c). [3] 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 the number 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.

[3]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	**Maximum** δ_g	**REP** %	**# of Plans**
QT5	1.03	32	6
QT8	1.05	25	6
AIQT5	1.03	32	20

Table 8: Effect of δ_g Setting (NodeExpand)

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 accommodate 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 also see in Table 3 that non-POSP plans do feature in the replacement plan diagrams, occasionally in significant proportions, as in 3DQT8 with RootExpand. Again, this phenomena is more prevalent in index-rich 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).

Effect of δ_g setting. As stated earlier, not all the replacements imposed by our algorithms are required for achieving stability. This is quantitatively highlighted in Table 8 where we show, for NodeExpand on a few representative query templates, the maximum value to which δ_g can be increased without compromising the stability achieved with the default setting of $\delta_g = 1$. We see here that the higher settings of δ_g result in the replacement percentage (REP%) coming down substantially from those listed in Table 7. As a specific case in point, running NodeExpand on AIQT5 with $\delta_g = 1.05$ achieves the same stability performance although the replacement percentage is brought down from 99% to only 32%. At first glance, these higher values may appear to be the preferred settings since they cause least disruption to the normal optimizer selections and are therefore more suitable from an industrial perspective. However, as shown quantitatively in Table 8, the higher δ_g settings have a side-effect – the plan diagram cardinalities may increase significantly. In fact, they can even result in a *loss of anorexia*, as observed with AIQT5 where the plan diagram cardinality jumps up to 20 from the 7 obtained in Table 7.

7.4 Computational Overheads

We now turn our attention to the price to be paid for providing plan stability and anorexic diagrams. The time aspect is captured in Table 9 where the per-query optimization times (in milliseconds) are shown for DP, RootExpand and NodeExpand – the increase relative to DP is also shown in parentheses. These results indicate that the performance of both replacement algorithms is within *10 to 20 milliseconds* of DP for all the templates.

With regard to memory overheads, shown in Table 10, the additional consumption is well within *10MB* (for RootExpand) and *100MB* (for NodeExpand) over all the query templates. These overheads

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 9: Time Overheads (in milliseconds)

appear quite acceptable given the richly-provisioned computing environments in vogue today. Further, this usage is incurred only for a very brief time period ($\ll 0.1s$), as per Table 9.

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 10: Memory Consumption (in MB)

Pruning Analysis. As presented in Section 4, our expansion algorithms involve a four-stage pruning mechanism, comprising of Cost, Safety, Benefit and Skyline checks. We show in Table 8, 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 (20%,20%) in S, we show the initial number of candidate wagons, and the number that remain after each check. As can be seen, there are over 250 plans at the beginning, but this number is pruned to less than five by the completion of the last check.

A large fraction of the overall pruning typically occurs due to the Cost-Safety-Benefit Skyline Check, as also seen in Table 8. We now show a visual example of this pruning quality through a Cost-Benefit skyline (which we have found to be a good approximation to the Cost-Safety-Benefit Skyline). Specifically, Figure 9 is a plot of all the plans (shown in red) input to the Cost-Benefit Skyline, while the green overlays are the ones that form the skyline of this plot. As can be seen from the example, over a hundred plans are pruned to a very small set of survivors, and this is a typical occurrence.

7.5 Efficacy of CornerAvg heuristic

In order to quantify the efficacy of the CornerAvg heuristic used by our algorithms, we also evaluated the AggSERF obtained through a "brute-force" algorithm, **OptimalAggSERF-SkylineUniversal**

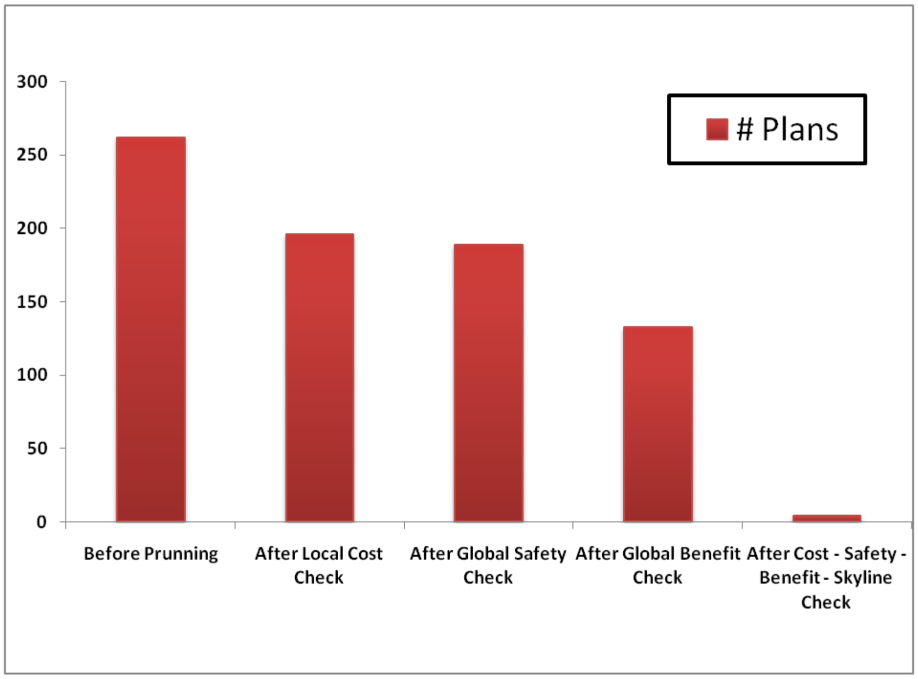


Figure 8: Impact of 4-stage Wagon Pruning

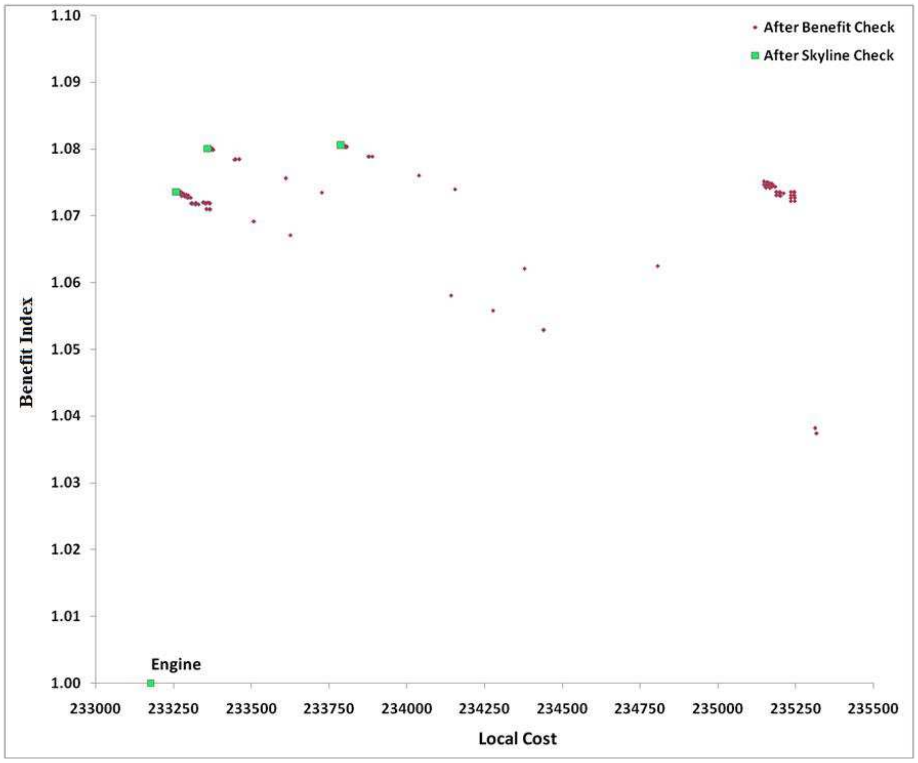


Figure 9: Cost-Benefit Skyline Pruning Example

(OAS-SU). OAS-SU explicitly and exhaustively checks for each query location, the best replacement with regard to the AggSERF metric, from the SkylineUniversal set of plans at that location. The performance of OAS-SU is showcased in Table 11 against that of NodeExpand and SkylineUniversal for all the query templates where SkylineUniversal was feasible.

The results of Table 11 are very encouraging since they demonstrate that the AggSERF achieved through CornerAvg *approaches that obtained with OAS-SU*, testifying to the potency of the CornerAvg heuristic. For example, on template 3DQT10, CornerAvg achieves an AggSERF of 0.39 as compared to OAS-SU's 0.44.

Query Template	**NodeExpand**		**SkyLineUniv**		**OAS-SU**	
	Rep %	**Agg SERF**	**Rep %**	**Agg SERF**	**Rep %**	**Agg SERF**
QT5	85	0.54	85	0.54	85	0.64
QT10	98	0.21	98	0.21	99	0.26
3DQT10	99	0.39	99	0.39	94	0.44
DSQT7	93	0.28	93	0.28	99	0.28
DSQT26	30	0.49	30	0.49	99	0.49

Table 11: AggSERF efficacy of CornerAvg heuristic

7.6 Performance with CC-SEER

As mentioned previously in Section 3, the CC-SEER algorithm guarantees global safety, unlike LiteSEER, which is a heuristic. A sample result where the safety aspect of CC-SEER is clearly evident is shown in Table 12, obtained by executing NodeExpand on query template AIQT5. [4] We see here that LiteSEER replacements resulting in negative MinSERF values, which go upto **-4.8**, are prevented by CC-SEER.

Query Template	**NodeExpand (LiteSEER)**				**NodeExpand (CC-SEER)**			
	Rep %	**Agg SERF**	**Min SERF**	**Harm %**	**Rep %**	**Agg SERF**	**Min SERF**	**Harm %**
AIQT5	94	0.91	-4.8	2%	93	0.96	0.0	0

Table 12: Guaranteed Replacement Safety with CC-SEER

The safety guarantee of CC-SEER is achieved at a price of increased computational overheads, and these overheads are shown in Table 13 for a representative set of templates. We see here that the time overheads of CC-SEER are 4 and 8 times that of LiteSEER for 2D and 3D templates, respectively. The space overheads are also higher for CC-SEER since each sub-plan has to now carry a larger number of corner costs to the higher levels, and this factor increases exponentially with dimensionality.

[4]This experiment was carried out on a commercial query optimizer since high negative MinSERF values did not arise on PostgreSQL with NodeExpand on any of our templates.

Query Template	NodeExpand (LiteSEER)		NodeExpand (CC-SEER)	
	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
QT5	18.9	6.2	81.5	15.9
QT10	4.8	3.9	20.4	5.4
3DQT8	26.4	29.5	215.3	118.1
AIQT5	20.9	11.8	86.1	31.5

Table 13: Computational Overheads of CC-SEER

7.7 Higher-dimensional Query Templates

In the previous experiments, the query templates were all either two or three-dimensional. Inherently, there is no fundamental restriction on applying our algorithms to higher-dimensional templates. However, from a practical viewpoint, there are two issues: Firstly, given a d-dimensional template, FPC costing has to be carried out at 2^d points with LiteSEER (as mentioned previously in Section 3). Therefore, the safety computation costs increase exponentially with dimensionality. Even with this increase, if we assume that query optimization times upto 1 second are acceptable, then it usually practical to produce replacement plan choices with as many as 6 dimensions in the selectivity space – a sample instance is shown in Table 14, where the overheads of optimizing query Q8 of the TPC-H benchmark using NodeExpand are evaluated with 4, 5 and 6 error-sensitive relations, respectively. We see here that NodeExpand takes about a quarter-second to optimize the 6D query, utilizing about 200MB of memory. We expect that such dimensionalities would prove sufficient in practice especially given that not all base relations would be sensitive to selectivity errors.

Dimen-siona-lity	DP		NodeExpand	
	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
4D			57	70
5D	6.0	4	119	113
6D			247	157

Table 14: Computational Overheads with Dimensionality

Secondly, explicitly demonstrating that our replacement algorithms do perform better on the various quality metrics as compared to the classical DP approach requires computing SERF values at all points of the selectivity space over all replacements. It is this *"proving"* process that is computationally time-consuming, not the plan generation process itself, and is the main reason for the limitation to 2D and 3D templates in our study. However, for users who unilaterally subscribe to the Expand approach, plans can be easily provided for higher-dimensional templates as well.

We have given sample performance results of NodeExpand for a 4D query template 4DQT8, in Table 15, for which it was feasible to complete the proving process with a low-resolution diagram. The SEER numbers are also given for comparative purposes.

Metric	NodeExpand	SEER	DP
Rep %	55 %	53 %	
AggSERF	0.19	0.21	
Help %	16 %	21 %	
MinSERF	0.0	0.0	
# of Plans	3	3	32
Non-POSP	1		

Table 15: NodeExpand on 4DQT8

8 Related Work

The effective handling of selectivity estimation errors has been a long-standing problem in the database literature. One approach has been to improve the quality of the statistical meta-data, for which several techniques have been presented including including refined summary structures [1], feedback-based adjustments [23, 8], hinting frameworks [7], and on-the-fly reoptimization of queries [17, 19, 3]. A complementary and conceptually different approach has been the identification of robust plans – that is, 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 [18].

Over the last decade, a variety of compile-time strategies have been proposed for identifying robust plans, including the Least Expected Cost [9, 10], Robust Cardinality Estimation [2] and Rio [3, 4] approaches. These techniques provide novel and elegant formulations, but, as described previously in [13], 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 [2, 3].

For completeness, we recapitulate from [13] a more detailed overview of the above compile-time strategies (a recent survey of run-time strategies is available in [11]): In the Least Expected Cost (LEC) approach [9, 10], it is assumed that the distribution of predicate selectivities is apriori available, and then the plan that has the least-expected-cost over the distribution is chosen for execution. While the performance of this approach is likely to be good on average, it could be arbitrarily poor for a specific query as compared to the optimizer's optimal choice for that query. Moreover, it may not always be feasible to provide the selectivity distributions.

An alternative Robust Cardinality Estimation (RCE) strategy proposed in [2] is to model the selectivity dependency of the cost functions of the various competing plan choices. Then, given a user-specified "confidence threshold" T, the plan that is expected to have the *least upper bound* with regard to cost in T percentile of the queries is selected as the preferred choice. The choice of T determines the level of risk that the user is willing to sustain with regard to worst-case behavior. Like the LEC approach, this too may be arbitrarily poor for a specific query as compared to the optimizer's optimal choice.

In the (initial) optimization phase of the Rio approach [3, 4], a set of uncertainty modeling rules from [17] are used to classify selectivity errors into one of six categories (ranging from “no uncertainty” to “very high uncertainty”) based on their derivation mechanisms. Then, these error categories are converted to hyper-rectangular error boxes drawn around the optimizer’s point estimate. Finally, if the plans chosen by the optimizer at the corners of the principal diagonal of the box are the same as that chosen at the point estimate, then this plan is assumed to be robust throughout the box.

However, in our framework, the above box essentially turns out to be the entire selectivity space and it is very unlikely that the plans chosen along the principal diagonal would be the same with respect to each other, let alone that at the point estimate. Therefore, it would be hard to obtain positive results for robustness. In contrast, our approach is to invoke plan replacement from a global perspective using the aggregate behavior over the corners of the selectivity space as indicators.

Both our previous offline SEER technique, and the online algorithms proposed in this paper, 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 paper (Section 4.2), the most important being, of course, that we implement an online intra-optimizer approach that is based on individual query instances, and does not require any global information to be supplied apriori.

Finally, our plan replacement approach only attempts to address 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 [16], 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 [11] for addressing selectivity errors in the higher nodes of the plan tree.

9 Conclusions and Future work

We investigated the systematic introduction of global stability criteria in the cost-based DP optimization process, with a view to reducing 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 associated computational overheads. Our approach expands 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 replacements are forwarded, and overheads are minimized.

We implemented, in the PostgreSQL kernel, a variety of replacement algorithms that covered the spectrum of design tradeoffs, and evaluated them on benchmark environments. Our results showed that a significant degree of robustness can be obtained with relatively minor conceptual changes to current optimizers, especially those supporting a foreign-plan-costing feature. Among the replacement algorithms, **NodeExpand**, which propagates the user’s cost and stability constraints to the internal nodes of the DP lattice, proved to be an excellent all-round choice. It simultaneously delivered good stability, replacement safety, anorexic plan diagrams, acceptable computational overheads, and near-optimal sub-plans. The typical situation was that its plan replacements were often able to eliminate 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 time and memory resources.

We hope that the promising results presented here would encourage commercial database vendors to incorporate such stability considerations in their optimization framework. Our purely compile-time techniques can be used in conjunction with run-time re-optimization strategies, as well as plan caching frameworks such as Progressive parametric query optimization (PQO) [6], to minimize the number of different plans that have to be considered during their execution.

In our future work, we plan to investigate automated techniques for identifying customized assignments to the node-specific cost, safety and benefit thresholds in the Expand approach. Further, it would be interesting to extend our study to skewed distributions of error locations in the selectivity space.

Acknowledgements. This work was supported in part by generous research grants from Microsoft and Google.

References

[1] 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.

[2] 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.

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

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

[6] P. Bizarro, N. Bruno and D. DeWitt, "Progressive Parametric Query Optimization", *IEEE TKDE*, 21(4), April 2009.

[7] N. Bruno, S. Chaudhuri and R. Ramamurthy, "Power Hints for Query Optimization", *Proc. of 25th IEEE Intl. Conf. on Data Engineering (ICDE)*, March 2009.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1986.

[19] 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.

[20] 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.

[21] 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.

[22] 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

[23] 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.

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

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

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

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

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

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

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

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

APPENDIX

A Proof of Skyline Sufficiency

In Section 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
$w1$	$c(p_{w1}, q_e) + \Delta_e$	$c(p_{w1}, V_i) + \Delta_{V_i}$	$c(p_{w1}, V_i) + \sum \Delta_{V_i}$
$w2$	$c(p_{w1}, q_e) + \Delta_e$	$c(p_{w2}, V_i) + \Delta_{V_i}$	$c(p_{w2}, V_i) + \sum \Delta_{V_i}$

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 arbitrary 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. ■

B Query Templates

We give below the specific query templates, based on the TPC-H and TPC-DS benchmarks, used in our experimental study. The bold-faced predicates correspond to the selectivity space dimensions.

B.1 TPC-H Query Templates

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

Figure 10: QT5 - 2D

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

Figure 11: QT8 - 2D

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

Figure 12: QT10 - 2D

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

Figure 13: QT8 - 3D

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

Figure 14: QT10 - 3D

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

Figure 15: QT8 - 4D

B.2 TPC-DS Query Templates

```
select
    i_item_id,
    avg(ss_quantity) as agg1,
    avg(ss_list_price) as agg2,
    avg(ss_coupon_amt) as agg3,
    avg(ss_sales_price) as agg4
from
    store_sales, customer_demographics,
    date_dim, item, promotion
where
    ss_sold_date_sk = d_date_sk
    and ss_item_sk = i_item_sk
    and ss_cdemo_sk = cd_demo_sk
    and ss_promo_sk = p_promo_sk
    and cd_gender = ’M’ and cd_marital_status = ’S’
    and cd_education_status = ’College’
    and (p_channel_email = ’N’ or p_channel_event = ’N’)
    and d_year = 2000
    and ss_sales_price :varies
    and i_current_price :varies
group by
    i_item_id
order by
    i_item_id
limit 100;
```

Figure 16: DSQT7

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

Figure 17: DSQT18

```
select
    i_item_id, avg(cs_quantity) as agg1,
    avg(cs_list_price) as agg2,
    avg(cs_coupon_amt) as agg3,
    avg(cs_sales_price) as agg4
from
    catalog_sales, customer_demographics,
    date_dim, item, promotion
where
    cs_sold_date_sk = d_date_sk
    and cs_item_sk = i_item_sk
    and cs_bill_cdemo_sk = cd_demo_sk
    and cs_promo_sk = p_promo_sk
    and cd_gender = 'M' and cd_marital_status = 'S'
    and cd_education_status = 'College'
    and (p_channel_email = 'N' or p_channel_event = 'N')
    and d_year = 2000
    and cs_list_price :varies
    and i_current_price :varies
group by
    i_item_id
order by
    i_item_id;
```

Figure 18: DSQT26