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

M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, Jayant R. Haritsa*
Database Systems Lab, SERC/CSA
Indian Institute of Science, Bangalore 560012, INDIA

ABSTRACT

Predicate selectivity estimates are subject to considerable run-time variation relative to their compile-time estimates, often leading to poor plan choices that cause inflated response times. We present here a parametrized family of plan generation and selection algorithms that replace, 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. These 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 algorithms is often competitive with those obtained through apriori knowledge of the plan search and optimality spaces. The additional computational overheads incurred by the replacement approach are miniscule in comparison to the expected savings in query execution times. 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

Most modern database query optimizers choose their execution plans on a cost-minimization basis. In this process, estimates of predicate selectivities are critical inputs to modeling the costs of query execution plans. Unfortunately, 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 Such errors arise due to a variety of reasons, including outdated statistics, attribute-value-independence (AVI) assumptions, and coarse summaries [14]. An adverse fallout of the estimation errors is that they often lead to poor choices of execution plans, resulting in grossly inflated query response times.

*Contact Author: haritsa@dsl.serc.iisc.ernet.in

Proceedings of the VLDB Endowment, Vol. 3, No. 1

Robust Plans. A variety of compile-time (i.e. optimization-time) strategies and run-time techniques have been proposed in the literature to mitigate the estimation problem. The particular approach we explore here is to identify, at compile-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* when faced with errors located *anywhere* in the 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 locally marginally costlier 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 [6].

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 (DP) exercise, employing 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 stabler 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.

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 stability criteria are invoked only 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 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) **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* [4]. The end result is that the root node of the DP lattice essentially receives the *entire plan search space*, modulo our wagon propagation heuristics; and (ii) **SEER** [8], 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 techniques have been implemented *inside* the PostgreSQL optimizer kernel, and their performance evaluated on a rich set of TPC-H and TPC-DS-based parametrized 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 within **100 milliseconds**, and memory overheads within **100MB**, RootExpand and NodeExpand often deliver plan choices that eliminate more than **two-thirds of the performance gap** (between P_{oe} and P_{oa}) for a non-trivial 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 with regard to the "exo-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* [7] 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, enumerated in detail in [7] – for example, it enhances the feasibility of parametric query optimization (PQO) techniques [9].

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 plan is usually the preferred 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, improving stability typically required replacing the plans for **30-50%** of the queries in the selectivity space, while additionally obtaining anorexic plan diagrams with NodeExpand required in excess of **80%** replacements.

To our knowledge, this is the first work to investigate the efficient identification of stable query execution plans with guaranteed local near-optimality and enhanced global stability.

2. PROBLEM FORMULATION

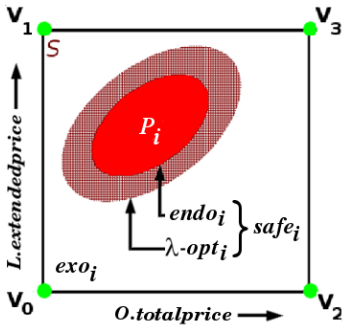
Consider the situation where the user has submitted a query and desires stability with regard to selectivity errors on one or more of the base relations that feature in the query. The choice of the relations could be based on user preferences and/or the optimizer's expectation of relations on which selectivity errors could have a substantial adverse impact due to incorrect plan choices. Let there be d such "error-sensitive relations" – treating each of these relations as a dimension, we obtain a d-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 – O and L – 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 fractional 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 later, 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 and run-time location is q_a, with P_{oe} and P_{oa} the optimal plans at these locations, respectively. 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 if 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,

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

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,

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

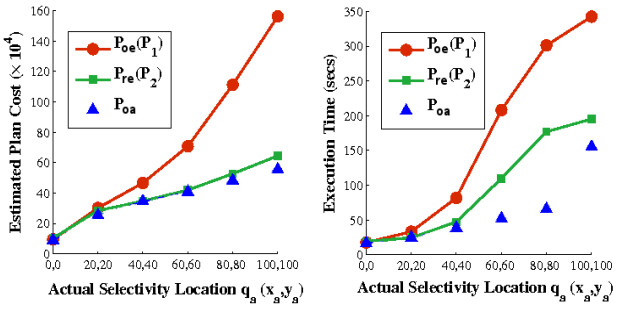
For example, setting $\lambda_g = 100\%$ stipulates that the cost of a query instance subject to plan replacement is guaranteed to be within twice its original value at all error locations in the selectivity space. We will hereafter refer to this constraint as *global-safety*.

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 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; its cost-optimal choice at the estimated location $(1\%, 40\%)$ 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, bordering on error "immunity", with respect to P_1.

To explicitly assess the compile-time predicted 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.



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

Figure 2: Benefits of Plan Replacement ($\hat{Q}10$, $\lambda_l, \lambda_g = 20\%$)

2.3 Error Resistance Metrics

Our quantification of the stability delivered through plan replacements is based on the **SERF** error resistance metric introduced in [8]. Specifically, for an error instance (q_e,q_a), the *Selectivity Error Resistance Factor* (SERF) of a replacement P_{re} wrt P_{oe} is computed as

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

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]

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

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 instances that could benefit from improved robustness.

Apart from AggSERF, we also compute **MinSERF** and **MaxSERF**, metrics 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.

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

   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.

[1] In [8], 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.

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 [12] 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 Appendix B.

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 Appendix A. 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" being 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, 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, we remove all wagons whose local cost significantly exceeds that of the engine. That is,

$$c(p_w, q_e) > (1 + \lambda_l^x)\ c(p_e, q_e) \tag{5}$$

where 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).

2. Global Safety Check: In the next step, we evaluate the behaviour of the "safety function", defined as

$$f(q_a) = c(p_w, q_a) - (1 + \lambda_g^x)c(p_e, q_a) \tag{6}$$

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 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 [8], 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 Appendix G).

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)** [13] 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 [13, 8] for the formal details).

We have also found that an extremely simple heuristic, called **LiteSEER** [8], which simply evaluates whether all the *corner points* are safe, that is,

$$\forall\ q_a \in Corners(\mathsf{S}),\ f(q_a) \leq 0 \tag{7}$$

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.

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

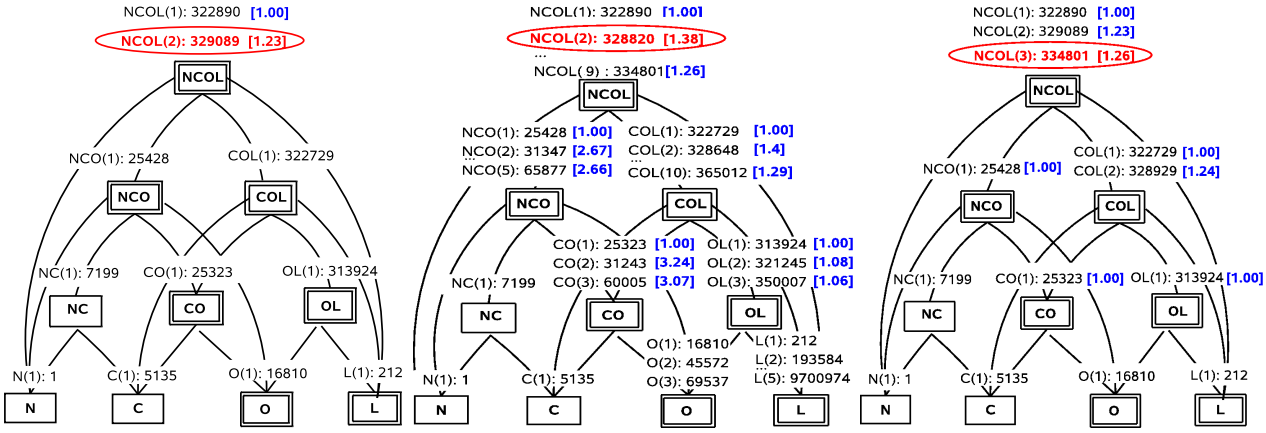
$$\xi(p_w, p_e) = \frac{\overline{c}(p_e, q_a)}{\overline{c}(p_w, q_a)} \qquad q_a \in Corners(\mathsf{S}) \tag{8}$$

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 with $\xi > 1$ 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 [7]. 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 (a more detailed analysis is given in Appendix C). 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 [4] of the wagons based on local cost, global safety and global benefit considerations. Specifically, for 2D error spaces, the skyline is comprised of five dimensions – the local cost and the four remote corner costs (the benefit dimension, when defined with the CornerAvg heuristic, becomes redundant since it is implied from the corner dimensions). A formal proof that the skyline-based wagon



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

Figure 3: Plan Expansion Algorithms ($\hat{Q}10$: $\lambda_l, \lambda_g = 20\%, \delta_g = 1$)

selection technique is equivalent to having retained the entire set of wagons is given in Appendix D.

After the above 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. Further, we also incorporate a benefit threshold, $\delta_g (\delta_g \geq 1)$, which determines the minimum benefit for which replacement is considered a worthwhile option. Ideally, δ_g should be set so as 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. The appropriate setting of δ_g is discussed in our experimental study (Section 5).

3.2 Plan Selection

At the end of the expansion process, a set of complete plans are available at the root node. There are two possible scenarios:
1) The only plan remaining is the standard cost-optimal plan P_{oe}, in which case this plan is output as the final selection; or
2) In addition to the cost-optimal plan, there are a set of candidate replacement plans available that are all expected to be more robust than P_{oe} (i.e. their $\xi > \delta_g$). To make the final plan choice from among this set, our current strategy is to simply use a *MaxBenefit* heuristic – that is, select the plan with the highest ξ.

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

A full-blown example of the plan replacement procedure is presented in Appendix E.

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. Alternatively, high values could be used at the leaves, progressively becoming smaller as we move up the lattice. Or, we could try exactly the opposite, with the leaves having low values and more relaxed thresholds going up the lattice. 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. The functioning of the algorithms is pictorially shown in Figure 3 for the example query $\hat{Q}10$ with $\lambda_l, \lambda_g = 20\%$ (and $\delta_g = 1$). In these figures, nodes that contain one or more error-sensitive relations (ORDERS, LINEITEM) in their sub-trees, are represented with double boxes.

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 are evaluated at the root node are filtered based on the four-check sequence, and a final plan selection is made from the survivors as per the procedure described in Section 3.2.

The functioning of RootExpand is pictorially shown in Figure 3(a), wherein the value above each node signifies the cost of the optimal sub-plan to compute the relational expression represented by the node – for example, the cheapest method of joining ORDERS (O) and LINEITEM (L) has an estimated cost of 313924. At the root node, the second-cheapest plan, NCOL(2), with cost 329089, is chosen in preference to the standard DP choice NCOL(1), due to locally being well within 20% of the lowest cost (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 error-sensitive nodes in the lattice interior, while the standard DP procedure is used at the remaining nodes. It represents the other end of the spectrum to RootExpand in that it propagates, beginning with the leaves, *all* wagons

evaluated at an error-sensitive 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 lattice interior. This implies that the root node effectively processes the *entire set of complete plans* present in the optimizer's search space for the query.

The pictorial representation of SkylineUniversal is shown in Figure 3(b). The labels above the error-sensitive nodes indicate the surviving wagons, along with their local costs and benefit indices. For example, CO(2) has a cost of 31243 and ξ = 3.24. The number of plans enumerated at the root node NCOL is 1099, and 9 of them successfully pass the four-stage check. The plan finally chosen is NCOL(2) 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 3(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 1 – standard DP is also included for comparative purposes.

Optimization Algorithm	Leaf Node λ_l^x, λ_g^x	Internal Node λ_l^x, λ_g^x	Root Node λ_l^x, λ_g^x	Root Node 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 1: Constraints of Plan Replacement Algorithms

Inheriting Engine Costs for Wagons. A crucial optimization incorporated in the above algorithms for reducing overheads is the following: 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.

4.1 Comparison with SEER

Our earlier SEER approach [8] identified robust plans through the *anorexic reduction of plan diagrams*. There are fundamental differences between that "offline/exo-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.
(iii) Finally, as previously mentioned, an attractive feature of NodeExpand is that it ensures performance fidelity of the replacement throughout its operator tree.

5. EXPERIMENTAL RESULTS

We implemented the above plan replacement algorithms in PostgreSQL 8.3.6 [15], operating on a Sun Ultra 24 workstation running Ubuntu Linux 9.10. The user-specified cost-increase thresholds in all our experiments was $\lambda_l, \lambda_g = 20\%$, a practical value as per our discussions with industrial development teams.

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 [1], are all based on queries appearing in the **TPC-H** and **TPC-DS** benchmarks, and cover both 2D and 3D selectivity spaces. 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 [16].

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 query locations where the optimizer's original choice is replaced; and **Help%**, the percentage of error instances wherein replacement plans reduced the performance gap substantially – specifically, by atleast *two-thirds*.

Replacement safety is evaluated through the MinSERF statistic and the percentage of error instances with MinSERF below $-\lambda_g$ is tabulated as **Harm%**.

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* [7, 8]. We also tabulate the number of *non*-POSP plans selected by our techniques.

3. Computational Overheads. This metric computes the overheads incurred, with regard to both time and space, relative to those experienced with the standard DP-based query optimization.

Query Template Descriptors. 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, while a label prefix of **3D** indicates a 3D template. The default physical design is a clustered index on each relation's primary key. Additional results obtained on an "index-rich" situation, denoted with label prefix **AI**, where indices exist on all query-related schema attributes, are given in Appendix F.1.

5.1 Plan Stability Performance

The stability performance results of the RootExpand, NodeExpand, SkylineUniversal and SEER algorithms are enumerated in Table 2 for a representative set of query templates from our study, which covered a spectrum of error dimensionalities, benchmark

Query Template	RootExpand REP %	RootExpand Agg SERF	RootExpand Help %	RootExpand # of Plans	RootExpand Non-POSP	NodeExpand REP %	NodeExpand Agg SERF	NodeExpand Help %	NodeExpand # of Plans	NodeExpand Non-POSP	SkylineUniversal REP %	SkylineUniversal Agg SERF	SkylineUniversal Help %	SkylineUniversal # of Plans	SkylineUniversal Non-POSP	SEER REP %	SEER Agg SERF	SEER Help %	SEER # of Plans	DP # of Plans
QT5	84	0.54	55	3	0	85	0.54	55	3	0	85	0.54	55	3	0	47	0.61	64	2	11
QT10	32	0.20	19	7	1	98	0.21	20	3	0	98	0.21	20	3	0	37	0.21	20	2	15
3DQT8	47	0.17	8	22	17	69	0.18	10	3	0	–	–	–	–	–	59	0.17	9	2	43
3DQT10	15	0.37	41	12	2	99	0.39	44	5	1	99	0.39	44	5	1	24	0.38	41	3	30
DSQT7	93	0.28	28	3	1	93	0.28	28	2	1	93	0.28	28	2	1	46	0.28	28	2	12
DSQT26	30	0.48	50	9	7	30	0.49	50	2	1	30	0.49	50	2	1	29	0.49	49	2	13

Table 2: Plan Stability and Plan Diagram Performance

databases, physical designs and query complexities (the complete set of results is available in [1]).

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 2 that REP% for both RootExpand and NodeExpand is quite substantial, even reaching in *excess of 90%* for some templates (e.g. DSQT7)! On average across all the templates, the replacement percentage was around 40% for RootExpand and 80% for NodeExpand.

We hasten to add that not all of these replacements are required to achieve stability, and the stability-superfluous replacements could be eliminated by setting higher values of δ_g. For example, with QT5, setting $\delta_g = 1.03$ achieves the same stability as the default $\delta_g = 1.0$ 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 contribute to producing 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: First, the Help% statistics indicate that, for several templates, a significant fraction of the error instances *do receive substantial assistance*. For example, QT5 has the performance gap reduced by more than 2/3 in about 55 percent cases, and, in fact, most of these receive SERF in excess of 0.9 – i.e., effectively achieve *immunity* from the errors. A visualization of the distribution of SERF values for this template is shown in Appendix F.3.

Second, the AggSERF performance of (offline) SEER is quite similar to that of RootExpand and NodeExpand. In our prior study [8], 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 SkylineUniversal ran out of memory are shown with –). In fact, as shown in Appendix F.2, 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.

Taken in toto, these results suggest that the controlled expansion technique is capable of extracting most of the benefits obtainable through plan replacement. Further, we have also conducted an analysis of the characteristics of the replacement plans vis-a-vis the original choices. Our observations, detailed in [1], indicate that (a) index-intersection joins are often replaced by scan-based joins; (b) nested-loop-based plans are frequently replaced with hash-join-based plans, while merge joins are almost never retained; and (c) left-deep plans are typically replaced by bushy plans.

Plan Replacement Safety. The MinSERF results with a LiteSEER implementation (given in Appendix F.5) indicate that this heuristic works very effectively in providing replacement safety since (a) only a few templates have negative MinSERF values, with small magnitudes, and (b) the harmful replacements in these cases occur for only a miniscule percentage of error locations. The corresponding CC-SEER results are given in Appendix F.6.

5.2 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 2, and to place them in context, the statistics for the standard DP-based optimizer are included.

Plan Diagram Cardinality. We see in Table 2 that for templates such as 3DQT8, where DP generates "dense" diagrams with high plan cardinalities, RootExpand diagrams may also feature a large number of plans. This behavior is more prevalent in index-rich environments (see Appendix F.1), 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 by DP, RootExpand and NodeExpand are shown in [1].

Non-POSP plans. We also see in Table 2 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 (see Appendix F.1) – 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.

5.3 Computational Overheads

We now turn our attention to the computational price to be paid for providing robust plans and anorexic plan diagrams. The time aspect is captured in Table 3 where the average 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 well within *100 milliseconds* of DP for all the templates.

Query Template	Optimization Time (ms)				
	DP	RootExpand		NodeExpand	
QT5	3.2	11.4	(+8.2)	22.2	(+19.0)
QT10	0.9	2.3	(+1.4)	3.2	(+2.3)
3DQT8	3.5	12.9	(+9.4)	30.6	(+27.1)
3DQT10	0.9	3.4	(+2.5)	4.3	(+3.4)
DSQT7	1.3	4.2	(+2.9)	7.7	(+6.4)
DSQT26	1.4	4.1	(+2.7)	7.0	(+5.6)

Table 3: Time Overheads (in milliseconds)

Query Template	Memory Overhead (MB)				
	DP	RootExpand		NodeExpand	
QT5	2.8	4.0	(+1.2)	7.0	(+4.2)
QT10	2.2	2.6	(+0.4)	3.4	(+1.2)
3DQT8	4.0	5.4	(+1.4)	10.6	(+6.6)
3DQT10	2.2	3.0	(+0.8)	5.1	(+2.9)
DSQT7	2.4	2.9	(+0.5)	3.5	(+1.1)
DSQT26	2.4	3.0	(+0.6)	3.8	(+1.4)

Table 4: Memory Consumption (in MB)

With regard to memory overheads, shown in Table 4, the peak additional consumption of RootExpand and NodeExpand is comfortably less than *100MB* over all the query templates. These overheads appear quite acceptable given the richly-provisioned computing environments in vogue today. Further, they are incurred only for a brief time period ($< 0.1s$), as per Table 3.

The low overheads are primarily due to the four-stage pruning mechanism that controls the number of wagons forwarded from a node – an example scenario is shown in Appendix F.4, where the initial 446 candidate plans are pruned to just 6 survivors.

6. RELATED WORK

Over the last decade, a variety of compile-time strategies have been proposed for identifying robust plans, including the Least Expected Cost [5], Robust Cardinality Estimation [2] and Rio [3] approaches. These techniques provide novel and elegant formulations, but, as described in detail in [8], 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].

Both our previous offline exo-optimizer SEER technique, and the online intra-optimizer 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 Section 4.1).

7. 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 reduce, by more than two-thirds, the adverse impact of selectivity errors for a significant number of error situations, in return for investing relatively minor time and memory resources.

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 partially supported by grants from Microsoft Research. We thank A. Dutt for implementation contributions.

8. REFERENCES

[1] M. Abhirama et al, "Stability-conscious Query Optimization", Tech. Rep. TR-2009-01, DSL/SERC, Indian Inst. of Science, July 2009. *dsl.serc.iisc.ernet.in/publications/report/TR/TR-2009-01.pdf*

[2] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach", Proc. of SIGMOD 2005.

[3] S. Babu, P. Bizarro and D. DeWitt, "Proactive Re-Optimization", Proc. of SIGMOD 2005.

[4] S. Borzsonyi, D. Kossmann and K. Stocker, "The Skyline Operator", Proc. of ICDE 2001.

[5] F. Chu, J. Halpern and P. Seshadri, "Least Expected Cost Query Optimization: An Exercise in Utility", Proc. of PODS 1999.

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

[7] Harish D., P. Darera and J. Haritsa, "On the Production of Anorexic Plan Diagrams", Proc. of VLDB 2007.

[8] Harish D., P. Darera and J. Haritsa, "Identifying Robust Plans through Plan Diagram Reduction", Proc. of VLDB 2008.

[9] A. Hulgeri and S. Sudarshan, "AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions", Proc. of VLDB 2003.

[10] N. Kabra and D. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans", Proc. of SIGMOD 1998.

[11] V. Markl et al, "Robust Query Processing through Progressive Optimization", Proc. of SIGMOD 2004.

[12] P. Selinger et al, "Access Path Selection in a Relational Database System", Proc. of SIGMOD 1979.

[13] H. Shrimal, "Characterizing Plan Diagram Reduction Quality and Efficiency", ME Thesis, Indian Inst. of Science, June 2009. *dsl.serc.iisc.ernet.in/publications/thesis/harsh.pdf*

[14] M. Stillger, G. Lohman, V. Markl and M. Kandil, "LEO, DB2's LEarning Optimizer", Proc. of VLDB 2001.

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

[16] *dsl.serc.iisc.ernet.in/projects/PICASSO/picasso.html*

APPENDIX

A. THE EXPAND ALGORITHM

The complete pseudo-code for the EXPAND family of algorithms, including the extensions described in Appendix B, is presented in Figure 4.

B. QUERY COMPLEXITIES

For ease of presentation, we had assumed in the main paper (Section 3) that optimizing the user query did not involve 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 appears above the join root node of the DP lattice). We now discuss the algorithmic extensions necessary to handle these features.

B.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 an *array* of sub-trains, one 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 in the lattice that combine 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 the 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.

B.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.

RootExpand. As explained in Section 4, plan expansion in RootExpand takes place only at the terminal node of the DP-lattice. This is appropriate when the terminal node 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 4.

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 1)
λ_g^x : Global-safety threshold for node x (set as per Table 1)
δ_g : Global-benefit threshold (set as per Table 1)

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 *io*}
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 io in $x.PlanTrain$ **do**
22: Move p to sub-train $x.PlanTrain_{io}$.
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 ξ in $x.PlanTrain$
54: Return $x.PlanTrain$

Figure 4: Node Expansion Procedure

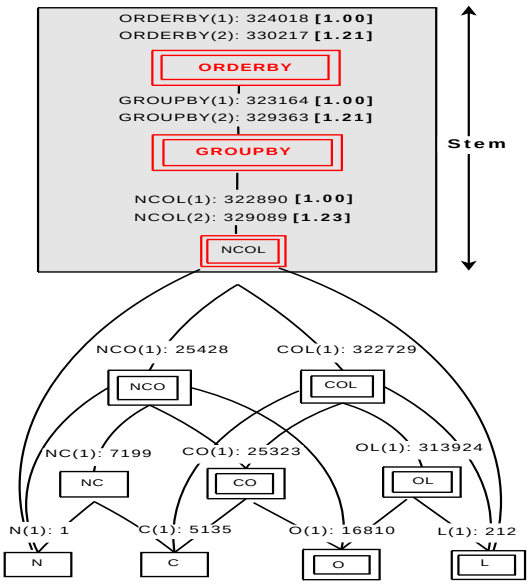


Figure 5: Plan Stem

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.

C. IMPACT OF PLAN REPLACEMENT

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 (with reference to 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. (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 λ_l, the local cost-bounding constraint. By virtue of this 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 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 study.

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.

D. PROOF OF SKYLINE SUFFICIENCY

In Section 3, we described a four-stage wagon 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 $LocalCost$, $Cost(V_1)$, $Cost(V_2)$, $Cost(V_3)$, $\ldots Cost(V_{2^n} - 1)$, $BenefitIndex$.

We now assess the relationship that develops between p_{w1} and p_{w2} in the event that both had 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. □

E. PLAN REPLACEMENT EXAMPLE

To make the plan replacement procedure concrete, consider the example situation shown in Table 5, 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).

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	1.23	
P10	329100	208219	230777	356567	1280674	1.23	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 5: Example Replacement at Root Node ($\hat{Q}10$)

F. ADDITIONAL RESULTS

Our experimental study covered a spectrum of error dimensionalities, benchmark databases, physical designs and query complexities. We present here additional experimental results relevant to the discussion in the main paper. The complete set of results is available in [1].

F.1 All Index Physical Configuration

In addition to the default physical design configuration, we considered **AllIndex (AI)**, an "index-rich" situation with (single-column) indices available on all query-related schema attributes. Representative results for the AI configuration are presented in Table 6 for replacement plan stability, and in Table 7 for plan diagram characteristics.

Query Template	RootExpand			NodeExpand			SEER		
	REP %	Agg SERF	Help %	REP %	Agg SERF	Help %	REP %	Agg SERF	Help %
AIQT5	87	0.37	36	99	0.37	38	87	0.38	39
AI3DQT8	30	0.18	21	98	0.19	21	55	0.12	15
AIDSQT18	11	0.03	1	75	0.07	3	68	0.04	3

Table 6: Plan Stability Performance (All Index)

We see in Table 6 that the stability results are, for the most part, qualititatively similar to those seen in the default primary-key-index scenario (Section 5). A point to observe here is that there are templates such as AIDSQT18, where the AggSERF values are extremely low. However, this appears to be an artifact of the database environment in which the evaluation was done rather than a basic flaw in our approach since even the yardstick algorithms, SEER and SkylineUniversal, are unable to achieve useful improvements on these templates. Moreover, as previously mentioned in Section 5, the SERF values obtained by SEER for the same templates were significantly higher on a commercial optimizer that offered a richer replacement space. Therefore, our expectation is that implementing the online algorithms in such high-end optimizers would result in a larger body of templates receiving significant AggSERF and Help% benefits.

Turning our attention to the plan diagram statistics in Table 7, we see that the observations made in Section 5 are more prominently portrayed here. Specifically, RootExpand features large plan cardinalities, whereas NodeExpand is comparatively anorexic. Further, non-POSP plans comprise a significant fraction of the plans appearing in the diagrams of RootExpand and NodeExpand.

Query Template	DP Plans	RootExpand Plans	RootExpand Non-POSP	NodeExpand Plans	NodeExpand Non-POSP	SEER Plans
AIQT5	29	13	3	7	4	4
AI3DQT8	70	51	41	14	12	7
AIDSQT18	28	31	7	3	1	3

Table 7: Plan Diagram Performance (All Index)

F.2 Efficacy of CornerAvg heuristic

In order to quantify the efficacy of the CornerAvg heuristic used by the Expand algorithms, we also evaluated the AggSERF obtained through a "brute-force" algorithm, **OptimalAggSERF-SkylineUniversal (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 8 against that of NodeExpand and SkylineUniversal for all the query templates of the main paper where SkylineUniversal was feasible.

The results of Table 8 are very encouraging since they indicate 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 the 0.44 of OAS-SU.

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 8: AggSERF efficacy of CornerAvg

F.3 Distribution of SERF values

A sample frequency distribution of the positive SERF values obtained with NodeExpand on QT5, which has a substantial Help% of over 50%, is shown in Figure 6. What is particularly notetworthy is that, by virtue of the plan replacements, a significant number of error instances essentially receive "immunity" (SERF ≥ 0.9) from the ill-effects of their estimation errors.

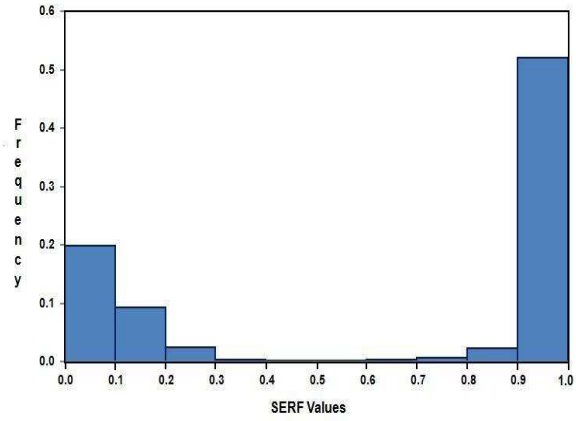


Figure 6: Frequency Distribution of SERF values (QT5)

F.4 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 9, a sample instance of the collective ability of these checks to reduce the number of wagons forwarded from a node to a limited viable number. In this table, obtained from the root node of a QT8 instance located at (30%,30%) in S, we show the initial number of candidate wagons, and the number that remain after each check. As can be seen, there are almost 450 plans at the beginning, but this number is pruned to less than 10 by the completion of the last check.

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

Table 9: Impact of 4-stage Wagon Pruning (QT8)

F.5 Plan Replacement Safety

We now shift our attention to the MinSERF metric to evaluate the *safety* aspect of plan replacement. To make sure that the replacements do not end up causing any material harm, MinSERF is calculated over the *entire selectivity space*. The results are presented in Table 10 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 through **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**.

Query Template	RootExpand		NodeExpand		SkylineUniversal	
	Min SERF	Harm %	Min SERF	Harm %	Min SERF	Harm %
QT5	0	0	0	0	0	0
QT10	-0.24	0.25	-0.24	0.01	-0.24	0.51
3DQT8	-1.05	0.01	-2.30	0.01	–	–
3DQT10	-1.08	1.93	-0.78	2.15	-0.78	2.15
DSQT7	0	0	0	0	0	0
DSQT26	0	0	0	0	0	0
AIQT5	0	0	0	0	–	–
AI3DQT8	-4.88	0.43	-2.80	4.30	–	–
AIDSQT18	0	0	0	0	–	–

Table 10: Plan Safety Performance

F.6 Performance with CC-SEER

As mentioned 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 11, obtained by executing NodeExpand on query template 3DQT8. We see here that LiteSEER replacements resulting in negative MinSERF values, which go upto **-2.3**, are prevented by CC-SEER.

Query Template	NodeExpand (LiteSEER)		NodeExpand (CC-SEER)	
	Min SERF	Harm %	Min SERF	Harm %
3DQT8	-2.30	0.01	0.0	0

Table 11: Guaranteed 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 12 for a representative set of templates. We see here that the time overheads of CC-SEER are substantially more than those of LiteSEER, the gap increasing with template dimensionality. 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.

Query Template	NodeExpand (LiteSEER)		NodeExpand (CC-SEER)	
	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
QT5	22.2	7.0	81.5	15.9
QT10	3.2	3.4	20.4	5.4
3DQT8	30.6	10.6	215.3	118.1

Table 12: Computational Overheads of CC-SEER

G. IMPLEMENTATION IN POSTGRESQL

We have implemented the various algorithms described in the previous section inside the PostgreSQL kernel, specifically version 8.3.6 [15]. 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 simplify 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.