On the stability of plan costs and the costs of plan stability

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

COMPUTER SCIENCE AND ENGINEERING

by

Abhijit P. Pai

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

July 2008

Acknowledgements

I'd like to thank

- Prof. Jayant Haritsa for being who he is to play a good role in making me who I am;
- my parents, for their unquestioning support and love;
- all my labmates and friends for their co-operation and help during the course of this work.

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 the first chapter of this report, a suite of 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 set 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 efficiently identify robust plan choices that substantially curtail the adverse effects of erroneous selectivity estimates.

The second chapter of this report deals with improvements made to Picasso and the PostgreSQL optimizer from Picasso's perspective. The Picasso tool produces various diagrams which enable the study of the behavior of a database query optimizer. Picasso is based on a client-server model. Both the client and server modules of this tool have been improved in many aspects. Server-side improvements include the betterment of the (compilation) plan diagram generation time estimator and porting of Picasso to IBM's Informix Dynamic Server database management system. Client-side enhancements include allowing for dynamic manipulation of Picasso user settings, saving diagram information into files, annotation of QTD's

depending upon the query execution type, regular expression search for the QTD list, saving of diagram visuals into the lossless .png format as well as other miscellaneous improvements. Some functionality that required interaction with the server and database engine were made client-only in order to speed up Picasso. All these enhancements have resulted in a much more useful and faster Picasso. The advanced free and open source database management system PostgreSQL's optimizer was augmented to produce and display multiple plans to the user. Also, remote costing was implemented in PostgreSQL.

Contents

List of Figures

Keywords

Query optimization, stable plans, selectivity error resistant plans.

Chapter 1

Introducing Stability in Cost-based Query Optimizers

1.1 Introduction

In modern database engines, the query optimizers choose their execution plans largely based on the classical System R strategy [16]: Given a user query, apply a variety of heuristics to restrict the combinatorially large search space of plan alternatives to a manageable size; estimate, with a cost model and a dynamic-programming-based processing algorithm, the efficiency of each of these candidate plans; 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, which can even be in orders of magnitude in real database environments [15], arise due to a variety of reasons [17], including outdated statistics, attribute-value independence assumptions and coarse summaries. An adverse fallout of these errors is that they often lead to poor plan choices, resulting in inflated query execution times.

Robust Plans. To address the above problem, an obvious approach is to improve the quality of the statistical meta-data, for which several techniques have been presented in the literature ranging from improved summary structures [1] to feedback-based adjustments [17] to on-the-fly reoptimization of queries [13, 15, 3]. A complementary and conceptually different approach, which we consider in this chapter, is to identify robust plans that are relatively less sensitive to such selectivity errors. In a nutshell, to "aim for resistance, rather than cure" by identifying plans that provide comparatively good performance over large regions of the selectivity space. Such plan choices are especially important for industrial workloads where global stability is as much a concern as local optimality [14].

Over the last decade, a variety of strategies have been proposed to identify robust plans, including the Least Expected Cost [5, 6], Robust Cardinality Estimation [2] and Rio [3, 4] approaches. These techniques provide novel and elegant formulations (summarized in Section 1.7). However, they 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, and on the other hand, neither have these techniques been shown to provide sustained good 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].

In this chapter, we present and evaluate a suite of plan generation and selection algorithms whose objective is to deliver substitute plan choices that are both (a) guaranteed to be locally near-optimal, and (b) likely to be globally stable, in comparison to the optimizer's solely cost-conscious choice. Of course, in some cases, the optimizer's plan may itself provide stable performance, in which case we retain it without substitution. An important criterion in the evaluation are the overheads involved in generating and identifying these substitute choices – if they are much larger than typical optimization costs, the database engine may be better off attempting a fresh optimization at run-time based on the encountered values, rather than

attempting to "second-guess" the optimizer at compile-time.

In essence, we investigate the design of a multi-metric (cost and stability) query optimizer in industrial-strength settings. Multi-metric considerations in optimizers is not a new concept – for example, PostgreSQL [19] supports using a combination of response time and latency to select execution plans. However, a critical difference in our work is the following: Our second metric, stability, is a *global* criterion whereas previous multi-metrics have been *local*, relevant only to the specific query instance under consideration.

Our proposed algorithms are based on judiciously expanding the candidate set of plan choices that are retained during the core dynamic-programming procedure, followed by a final selection heuristic that takes both cost and stability into account. These algorithms have been implemented in the PostgreSQL optimizer kernel and their performance has been 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 one of the selection algorithms, called **InternalFixed**, is indeed capable of efficiently making plan choices that substantially curtail the adverse effects of selectivity estimation errors. Specifically, while incurring overheads that are within a few multiples of the normal optimization time, it delivers plan choices that improve the selectivity error resistance by a factor of XX.

A valid question at this point would be whether in practice the optimizer's cost-optimal choice usually turns out to itself be the stable choice as well – that is, are optimizers inherently stable? Our experiments clearly demonstrate that this is not the case since the proportion of query points in the selectivity space for which plan substitution took place was quite large – in the range of 30%–80% for the environments considered in our study.

Comparison with Stability through Plan Diagram Reduction. In an earlier work [10], we had attempted to identify plans that are robust to selectivity errors through the following process: First generate "plan diagrams" [26], which are color-coded pictorial enumerations of the optimizer's plan choices over the selectivity space. Then, apply "anorexic reduction algorithms" [9, 10] to convert these diagrams to much simpler pictures featuring significantly fewer

plans, without materially degrading the processing quality of any individual query. The reduction process is usually highly successful in removing locally optimal plans that suffer from globally volatile behavior, and the few retained plans typically exhibit good error resistance characteristics.

There are some critical differences between this earlier "post-facto reduction" approach and our current "online production" work:

- We do not have the luxury of assuming that the entire plan diagram is apriori available when we optimize a given user query. Instead, our challenge is to identify, based on comparatively very limited knowledge, the appropriate plan choices for an individual query. That is, if all the queries in the selectivity space were to be individually optimized, then the resultant plan diagram produced by our techniques should ideally have similar characteristics to that of the post-facto reduced diagram.

- The set of plans in the plan diagram produced with our online approach could potentially include plans, unlike the post-facto approach, that are *outside* of the parametric optimal set of plans (POSP) [11]. This opens up the possibility, at least in principle, of obtaining even greater reduction than that obtained by the post-facto approach. For example, consider the situation wherein there is a very good plan that is always second-best by a small margin over the entire selectivity space. In this case, the post-facto approach would, by definition, not be able to utilize this plan, whereas it would certainly fall within the ambit of the plan candidate set in our online approach. This is confirmed in our experimental study wherein non-POSP plans do regularly feature in the set of recommended plans.

  On the other hand, our strategies, due to employing sub-plan pruning techniques for minimizing computational overheads, may fail to consider some POSP possibilities. Our experience thus far has been that the tradeoff is always in favor of the online approach.

- Because the post-facto approach comes into play after the plan diagram has been fully generated, it is able to provide guarantees about the global stability of its plan choices. In our case, however, it is not possible to do so without incurring unviably large overheads, and therefore we can only resort to empirical assessments to evaluate the stability of the

techniques. The silver lining is that our results show that the stability achieved is usually comparable to the post-facto approach.

- Finally, and most importantly, the post-facto approach, which deals with complete plans that have already been constructed, is able to address only selectivity errors that occur on the *base relations*. However, in the online approach, stability criteria can be applied at every stage of the plan generation process, that is, at the level of *sub-plans*. This means that stability considerations are incorporated at *all nodes* in the entire operator tree, not merely the leaves.

Contributions. In summary, we present a framework in this chapter to analyze the production of query execution plans that take into account both local optimality and global stability perspectives. The framework opens up a rich algorithmic design space, and we explore a part of this space here in the context of industrial-strength database environments. The initial results have turned out to be quite promising and we hope that they will trigger further investigation of this highly practical issue. 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. In fact, it may be even easier since most of these engines natively provide the "Foreign-Plan-Costing" feature which supports costing of plans outside of their optimality regions, whereas we have had to explicitly add this feature in the PostgreSQL optimizer.

Organization. The remainder of this chapter is organized as follows: in Section 1.2, we describe the overall problem framework and motivation. Our new set of plan selection algorithms are presented in Section 1.3, and strategies for minimizing their overheads are described in Section 1.4. Implementation issues related to incorporating these algorithms within the PostgreSQL codebase are narrated in Section 1.5. The experimental framework and performance results are highlighted in Section 1.6. Related work is overviewed in Section 1.7. Finally, in Section 1.8, we summarize our conclusions and outline future research avenues.

1.2 Problem Formulation

Consider the situation where the user has submitted a query and we would like to have stability with regard to selectivity errors on some or all 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 query shown in Figure 1.1(a), which is based on Query 10 of the TPC-H benchmark [23] – this query has four base relations (NATION, CUSTOMER, ORDERS, LINEITEM), two of which are deemed to be error relations (ORDERS, LINEITEM), symbolized by the double boxes in Figure **??**. For this query, the associated 2-D selectivity space S is shown in Figure 1.1(b).

For ease of presentation, we will assume hereafter that S is two-dimensional (our experiments in Section 1.6 consider 3-D spaces as well). Within S, each point $q(x, y)$ corresponds to a unique query with selectivities x, y in the X and Y dimensions, respectively. We use $c_i(q)$ to represent the estimated cost of executing a query point q with plan P_i.

1.2.1 Cost Constraint on Plan Replacement

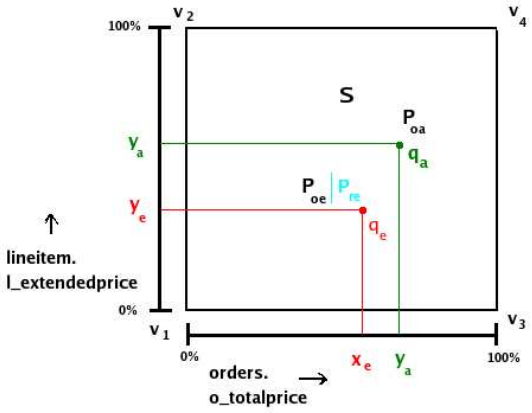
Consider a specific query point q_e, whose optimizer-estimated location in S is (x_e, y_e). Denote the optimal plan choice (as determined by the optimizer) at point q_e by P_{oe}. Now, given a user-defined maximum-cost-increase threshold λ ($\lambda \geq 0$), it is permissible to substitute P_{oe} with an alternative replacement choice P_{re}, only if

$$\frac{c_{re}(q_e)}{c_{oe}(q_e)} \leq (1 + \lambda) \tag{1.1}$$

For example, setting $\lambda = 10\%$ stipulates that the estimated cost of a query point subject to plan replacement is guaranteed to be within 1.1 times its original value. We will refer to this constraint hereafter as λ-optimality.

```
select *
    from customer, orders, lineitem,
nation
    where c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and c_nationkey = n_nationkey
    and o_totalprice ≤ 943.47
    and l_extendedprice ≤ 1279.13
```

(a) **Query Instance**



(b) **Selectivity Space**

Figure 1.1: **Sample Query and Selectivity Space (based on TPC-H Q10)**

1.2.2 Selectivity Estimation Errors

Due to errors in the selectivity estimates, the *actual* location of q_e could be different at execution-time – denote this location by $q_a(x_a, y_a)$, and the optimizer's optimal plan choice at q_a by P_{oa}. Now, given that the P_{oe} choice at q_e has been replaced by an alternative plan P_{re} due to stability considerations, the actual query point q_a will be located in one of the following disjoint regions of P_{re} that together cover S:

Endo-optimal region of P_{re}: Here, q_a is located in the optimality region of the replacement plan P_{re}, which also implies that $P_{re} \equiv P_{oa}$. Since $c_{re}(q_a) \equiv c_{oa}(q_a)$, it follows that the cost of P_{re} at q_a, $c_{re}(q_a) < c_{oe}(q_a)$ (by definition of a cost-based optimizer). Therefore, improved resistance to selectivity errors is always *guaranteed* in this region.

Note that, as mentioned earlier, unlike in [9, 10], P_{re} is not restricted to be only from the parametric optimal set of plans (POSP) over S, but in principle, could be *any plan* from the optimizer's search space that satisfies λ-optimality. If P_{re} is not from the POSP, then it will not have any endo-optimal region.

Replacement-region of P_{re}: Here, q_a is located in the region "swallowed" by P_{re}, replacing the optimizer's cost-optimal choices due to stability considerations. By virtue of the λ-threshold constraint, we are assured that $c_{re}(q_a) \leq (1 + \lambda)c_{oa}(q_a)$, and by implication that $c_{re}(q_a) \leq (1 + \lambda)c_{oe}(q_a)$. Now, there are two possibilities: If $c_{re}(q_a) < c_{oe}(q_a)$, then the replacement plan is again guaranteed to improve the resistance to selectivity errors. On the other hand, if $c_{oe}(q_a) \leq c_{re}(q_a) \leq (1 + \lambda)c_{oe}(q_a)$, the replacement is guaranteed to not cause any real harm, given the small values of λ that we consider in this chapter.

Exo-optimal region of P_{re}: Here, q_a is located outside both the endo-optimal and stability-regions of P_{re}. At such locations, we cannot apriori predict P_{re}'s behavior, and therefore the replacement may not always be a good choice – in principle, it could be arbitrarily worse. Establishing that a replacement is not a bad choice anywhere in S, while

technically feasible, incurs hugely unviable overheads, as explained later in the chapter. Therefore, we have to take recourse to heuristics instead – the silver lining is that, as shown subsequently in our experimental results, there do exist simple heuristics that are both efficient and mostly correct in their decisions.

1.2.3 Error Resistance Metric

To explicitly quantify the stability delivered through plan replacement, we use the Error Resistance Metric defined in [10]. This states that given an estimated query location q_e and an actual location q_a, the *Selectivity Error Resistance Factor* (**SERF**) of a replacement plan P_{re} w.r.t. the optimal plan P_{oe} is defined as:

$$SERF(q_e, q_a) = 1 - \frac{c_{re}(q_a) - c_{oa}(q_a)}{(1+\lambda)c_{oe}(q_a) - c_{oa}(q_a)} \tag{1.2}$$

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

The above formula applies to a specific instance of replacement. To capture the net impact of plan replacement on improving the resistance in the *entire space* S, we compute the following:

$$AvgSERF = \frac{\sum_{q_e \in rep(\mathsf{S})} \sum_{q_a \in exo_{oe}(\mathsf{S})} SERF(q_e, q_a)}{\sum_{q_e \in rep(\mathsf{S})} \sum_{q_a \in exo_{oe}(\mathsf{S})} 1} \tag{1.3}$$

where $rep(\mathsf{S})$ is the set of points in S that were replaced, and $exo_{oe}(\mathsf{S})$ is the set of points lying in the exo-optimal region defined with respect to P_{oe}, the optimizer's plan choice for q_e. The normalization is with respect to the number of possible selectivity errors in the diagram.

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 uniformly

distributed over this space. However, our conceptual framework is also applicable to the more generic case where the error locations have an associated probability distribution.

We also compute the metrics MinSERF and MaxSERF, the minimum and maximum values of SERF over all replacement instances. Values of MaxSERF that are close to the maximum value of 1 indicate that some replacements have provided immunity to specific instances of selectivity errors. On the other hand, negative values for MinSERF indicate that some replacements have been harmful. We measure the proportion of such harmful instances in our experiments.

1.2.4 Problem Definition

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

Stable Plan Selection Problem. Given a selectivity space S and maximum cost-increase-threshold λ, implement a plan substitution strategy such that:

1. $\forall q \in rep(\mathsf{S}), \frac{c_{re}(q)}{c_{oe}(q)} \leq (1+\lambda)$
2. MinSERF ≥ 0, and
3. AvgSERF is maximized.

The first criterion guarantees λ-optimality, the second assures that there are no harmful replacements, while the third captures the stability improvement objective.

1.3 Plan Selection Algorithms

In this section, we present a set of plan selection algorithms that attempt to address the Stable Plan Selection problem. Our algorithms 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.

For ease of presentation, we will assume that there are no "interesting order" plans [16] present in the search space – however, they are accounted for in our implementation. Further, while any cost metric is acceptable in principle, we will assume that the cost is indicative of the query response time.

There are two aspects to our algorithms: First, a procedure for the generation of candidate replacement plans, and second, a selection strategy to pick a stable replacement from among these candidates.

1.3.1 Generation of Candidate Replacement Plans.

In the normal dynamic-programming (DP) based exercise of building up the optimal plan for a query, the cheapest sub-plan at each node is propagated to the nodes at the higher levels. Finally, at the root of the DP tree, which signifies complete plans for the the entire query, the cheapest strategy is chosen as the execution plan by the optimizer. An example DP tree is shown in Figure 1.2, corresponding to the query of Figure 1.1(a). In this picture, the value above each node signifies the cost of the optimal sub-plan to compute the relational expression represented by the node – for example, the cheapest method of joining ORDERS (O) and LINEITEM (L) has an estimated cost of 312,593.

The Root Algorithm. Root, the first of our new algorithms, is a simple variant of the standard DP procedure, and is pictorially shown in Figure 1.3. Here, DP is used 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 first filtered into a λ-compliant group with respect to the cheapest plan. Subsequently, a stability criterion (discussed in the following sub-section) is employed to decide which of the candidate plans in the λ-group is selected to execute the query.

The filtration step at the root node ensures that Condition 1 of the Stable Plan Selection Problem is satisfied, and is also followed by all our other algorithms. Further, the generation overheads as compared to standard DP are minimal since the stability issue is only addressed in the final root node. However, by the same token, the size and diversity of the candidate

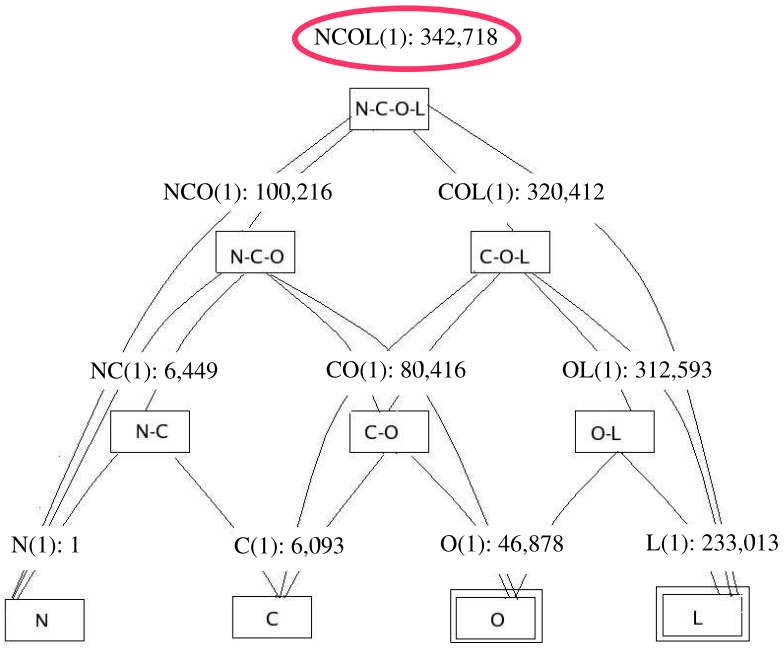


Figure 1.2: Dynamic Programming

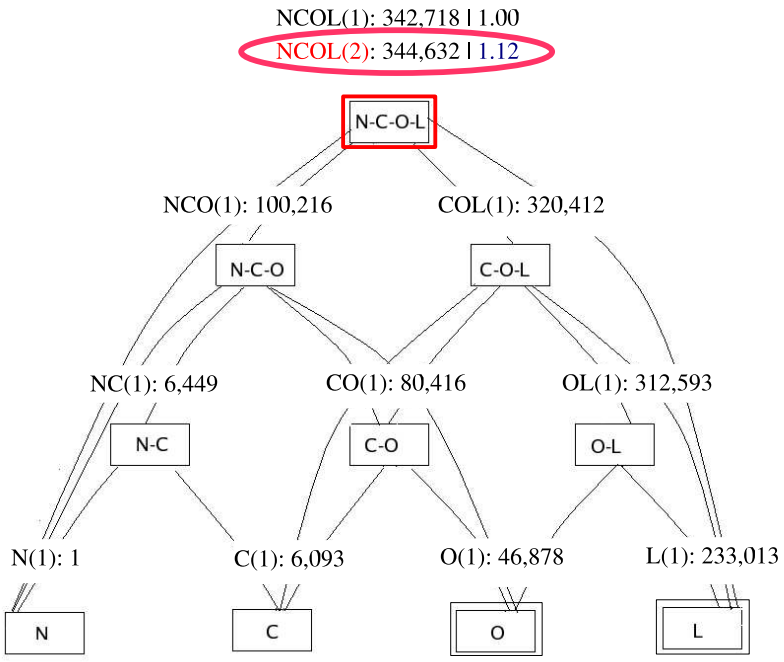


Figure 1.3: Root Algorithm ($\lambda = 10\%$)

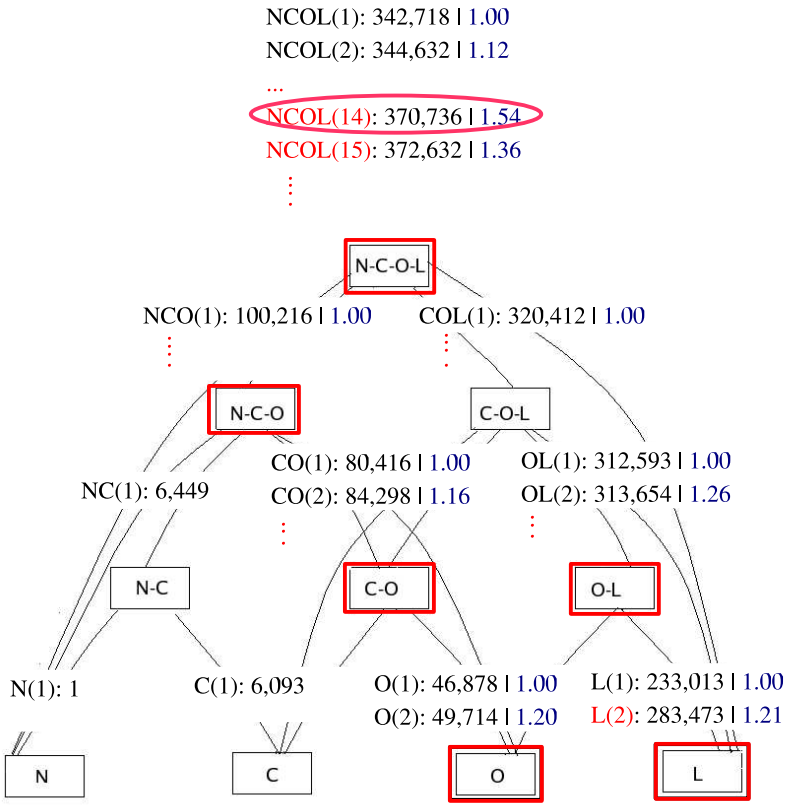


Figure 1.4: Universal Algorithm ($\lambda = 10\%$)

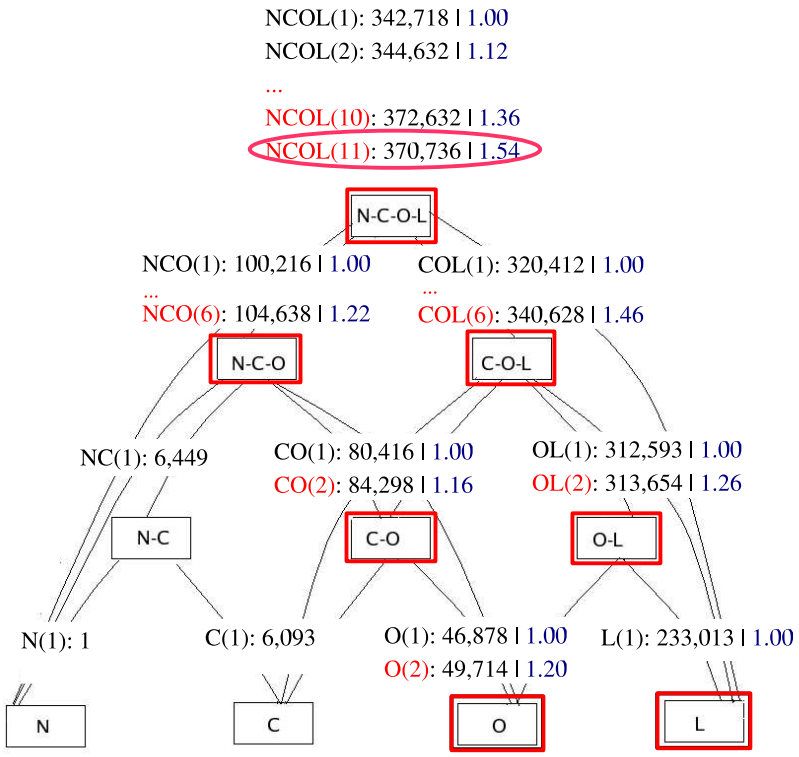


Figure 1.5: InternalFixed Algorithm ($\lambda = 10\%$)

replacement plan set is also extremely limited. This issue is addressed in our remaining algorithms which allow a richer set of replacement plans to reach the root node.

The Universal Algorithm. The Universal algorithm represents the other end of the spectrum to Root in that it propagates, beginning with the leaves, *all* sub-plans evaluated at a node to the levels above. That is, there is absolutely no pruning anywhere in the internals of the tree, resulting in the root node processing the *entire set of complete plans* present in the optimizer's search space for the query. A pictorial representation of Universal is shown in Figure 1.4.

Obviously, this approach represents the extreme with regard to maximizing the scope for finding replacement plans. However, as should be expected given the well-known exponential size of the search space, the overheads quickly become unviably large with increasing query complexity.

The InternalFixed Algorithm. The InternalFixed Algorithm, shown in Figure 1.5, strikes an intermediate balance between replacement richness and generation overheads, and represents the middle ground between Root and Universal. Specifically, it allows at each internal node, the cheapest sub-plan and all other plans that are within $(1 + \lambda_i)$ of its cost to be propagated to the next level. Note that λ_i is an algorithmic parameter and therefore its setting can be made independently of λ, the user's constraint, which is always applied at the final root node.

If λ_i is set to 0, the Root algorithm results, whereas $\lambda_i = \infty$ is equivalent to the Universal algorithm. Therefore, the choice of λ_i can be used to strike the desired tradeoff between these extremes. The results reported in this chapter are for λ_i set to 10%.

A large number of variations on the above algorithms are feasible. As a case in point, the λ_i setting could be a function of the individual nodes rather than a constant. For example, a high value of λ_i 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 λ_i 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, and our assessment here explores only a part of this space. We intend to study other algorithmic constructions in our

future work.

1.3.2 Replacement Plan Selection

We now turn our attention to the second component of our algorithms, namely the stability considerations determining plan selection from among the cost-compliant choices at the root node.

For each plan P_{re} in the candidate replacement set, we evaluate its StabilityIndex with the following simple equation which computes the inverse of the cumulative cost of the plan over the entire space S, normalized to the corresponding value for the cost-optimal plan:

$$StabilityIndex(P_{re}) = \frac{\sum_{q \in S} c_{oe}(q)}{\sum_{q \in S} c_{re}(q)} \tag{1.4}$$

Then, the plan with the maximum StabilityIndex value is selected as the desired plan. That is, our preferred choice is the plan that has the *lowest cumulative cost* over the entire space S. Note that the StabilityIndex values greater than 1 indicate a plan that is more stable than the cost-optimal plan, whereas values less than 1 indicate a less stable plan (the cost-optimal plan itself will always have, by definition, a StabilityIndex of 1).

Note that the above is only *one* characterization of stability, and it is entirely possible to contemplate alternate definitions. For example, in addition to the aggregate cost, we could also take into account the variance of the cost distribution. However, as our experimental results will demonstrate, the simple formulation of Equation 1.4 appears to work quite satisfactorily for the rich environments that we have analyzed.

Irrespective of the specific choice of stability index, an important factor to ensure is that the index should give the same ranking between a pair of plans *irrespective of the specific query* q_e *that is currently being optimized*. That is, the stability of a plan vis-a-vis another plan should be determined by its *global* behavior over the entire space.

Finally, even with the simple definition of Equation 1.4, computing StabilityIndex can be extremely expensive. This is because it requires evaluating the costs of all the candidate plans *over the entire selectivity space*. Therefore, we need to take recourse to efficient heuristics,

Algorithm	Internal Node		Root Node	
	λ_i	ξ_i	λ_r	ξ_r
Standard DP	0	–	0	–
Root	0	–	λ	Max(ξ_{re})
InternalFixed	> 0	> 1	λ	Max(ξ_{re})
Universal	∞	–	λ	Max(ξ_{re})

Table 1.1: Constraints of Plan Selection Algorithms

like the one described next.

Corner Heuristic. In this heuristic, the costs of the entire space are simply represented by the costs at the *corners* of the space, that is:

$$StabilityIndex(P_{re}) = \frac{\sum_{q \in Corners(\mathsf{S})} c_{oe}(q)}{\sum_{q \in Corners(\mathsf{S})} c_{re}(q)} \tag{1.5}$$

Specifically, in the 2-D case of Figure 1.1(b), the costs at the vertices V1(0,0), V2(0,100), V3(100,0) and V4(100,100) in S would be evaluated for all the candidate replacement plans.

The constraints imposed by the various algorithms presented above are summarized in Table 1.1, with ξ used to represent the StabilityIndex.

1.4 Optimizations for InternalFixed

As discussed in the previous section, the InternalFixed algorithm permits, in addition to the cheapest sub-plan at each node, sub-plans that satisfy the λ_i constraint to also be propagated to the upper levels. Due to the multiplicative nature of the DP tree, the computational overheads arising out of these additional sub-plans, if not carefully regulated, can quickly spiral out of control. In the remainder of this section, we describe a variety of optimizations that can be collectively used to restrict the overheads to acceptable levels. Two of the optimizations are theoretically justified based on sub-plan cost behavior, while the third is a worst-case heuristic.

For ease of understanding, we will use the term "train" to refer to the array of sub-plans being propagated from one node to another, with the "engine" being the cost-optimal sub-plan

and the "wagons" being an ordered sequence of the other candidate sub-plans. The ordering can be based on execution-cost or on stability-index, as described below.

Standard DP for Error-free Sub-Trees: The costs of sub-trees in the plan generation tree that do not feature any error-sensitive relations in their leaves are expected to not be materially impacted by selectivity errors. Therefore, the standard DP procedure wherein only the cheapest sub-plan is propagated upwards can be used within these sub-trees. This is the reason for only single plans being forwarded in the N-C sub-tree component in Figure 1.5 since the leaves – NATION and CUSTOMER – are not error-sensitive relations.

Reusing Engine Costs for Wagons: When two plan-trains arrive and are combined at a node, note that the costs of combining the engines of the two trains in a particular method is exactly the same cost as that of combining any other pair from the two trains. This is because the engines and wagons in any train all represent the same input data. Therefore, we need to only combine the two engines in all possible ways, just like in standard DP, and then simply reuse these associated costs to evaluate the total costs for all other pairings between the two trains.

Cap on Wagon Length: Even with the above two optimizations, it is possible that with certain queries, due to the presence of a large number of λ_i-compliant sub-plans, the plan-train may become rather long and result in a runaway situation with regard to overheads. Therefore, an absolute length-cap $(\hat{l})$ can be enforced at each node to prevent such situations. For example, a limit of $\hat{l} = 5$ sub-plans to be forwarded from each internal node is enforced in Figure 1.5.

Now, given a situation where the wagon-length exceeds $\hat{l}$, making the cap come into play, an issue that immediately arises is how should the $\hat{l}$ survivors be chosen? One obvious possibility is to retain the cheapest $\hat{l}$ plans, in the expectation that they would survive

until the root node. However, given our eventual stability objective and the fact that cost has already been taken into account using the λ_i constraint, a more attractive alternative is to choose the $\hat{l}$ plans based on their *stability indices*. This would explicitly implement our intuition that stable complete plans tend to be built from stable sub-plans. That is, we build a stability-based-train instead of a cost-based-train, and this notion is shown in Figure 1.5. By deliberate design, the cap operates only on the wagons while the engine (cost-optimal sub-plan) is always forwarded to the upper levels no matter what. This is because we would like to ensure that the normal DP-based plan always features in the final root node.

When all the above optimizations are included, our experience has been, as borne out quantitatively in the experimental results (Section 1.6), that the overall optimization time is always *within a few multiples* of the standard DP time for the benchmark queries. For example, for 2D selectivity spaces, the running time is usually within twice that of DP, while for 3D spaces, it is generally within or around three times that of DP. The important point to note here is that the savings in execution time due to selecting robust plans can far outweigh the additional effort spent in optimization.

An abstract version of the InternalFixed algorithm, incorporating all the above optimizations, is shown in Figure 1.6.

1.5 Implementation in PostgreSQL

We have implemented the various algorithms described in the previous section inside the PostgreSQL [19] kernel, specifically version 8.2.5 [20]. In this section, we discuss the issues related to our implementation experience.

1.5.1 Foreign Plan Costing.

In order to implement the CornerStability heuristic described in Section 1.3.2, we need to be able to cost, at all corners of S, the set of sub-plans featuring in the λ-set at a node. This

requires the underlying database engine to support the costing of *"foreign plans"*, that is, of costing plans in their *exo-optimal* regions. On the bright side, the foreign-plan-costing (FPC) feature has become available in the recent versions of several commercial optimizers, including DB2 [18] (Optimization Profile), SQL Server [21] (XML Plan) and Sybase [22] (Abstract Plan). However, on the down side, this feature is not available in PostgreSQL.

Forced Optimality. One option that we could consider to get around this lacuna in PostgreSQL is to force the plan whose cost we wish to evaluate at an exo-optimal location to *become* the optimal plan at that location – this could be attempted, for example, by employing the PostgreSQL feature that allows users to enable or disable operators like hash-join, merge-join, etc. However, this proves to be an extremely cumbersome and mostly infeasible scheme for the complex plans encountered in practice, especially since the enabling and disabling cannot be conditionally applied at selective locations in the plan-tree.

Universal Plan Generation. An alternative approach is to utilize the fact that there are only a few corner locations at which we wish to carry out FPC. Specifically, we could generate the *Universal* set of plans at these locations and amortize the associated overheads over all future instances of the user query. However, even this is not feasible since the universal set becomes unmanageably large once the query complexity goes beyond a few relations.

Plan-tree Costing. Therefore, we have taken the alternative tack of directly implementing remote costing in the PostgreSQL optimizer kernel. Our initial idea was to merely carry out a bottom-up traversal of the foreign-plan's operator tree 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 pure costing approach and we found significant discrepancies between our non-cached estimation of costs and the corresponding cache-based estimates provided by PostgreSQL.

Piggybacked Plan-tree Costing. To resolve the caching issue, we decided to perform *anticipatory* FPC at the desired locations *in parallel* with the local costing of each sub-plan during the optimization process. That is, we resorted to "piggy-backing" on the local cost computations. This solution does work out well but, on the down side, incurs unnecessary overheads since even those sub-plans that are later discarded in the plan generation process have to be remotely costed along the way.

1.5.2 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.

1.5.3 Coding Effort

The optimizer component of PostgreSQL 8.2.5 has five sub-modules which cumulatively contain close to 40000 lines of code. On this codebase, implementing the above-mentioned features required adding or modifying around 2500 lines of code in 650 locations.

1.6 Experimental Results

The algorithms were implemented on PostgreSQL 8.2.5 [20] on a Sun Ultra 20 workstation with an AMD Opteron dual core 2.5 Ghz processor, 4 GB of main memory and 500 GB of hard disk, running Redhat Enterprise Linux 9.

In order to validate our performance characteristics over the entire selectivity space, plan diagrams were produced using the Picasso query optimizer visualization tool [25] for our modified versions of the optimizer as well as a base version that uses the regular DP strategy for optimization.

A variety of two and three-dimensional **TPC-H**-based query templates (with uniformly

Algorithm InternalFixed ()
For each level of the DP search tree starting with level 1

If we are at the topmost level of the plan tree, set λ equal to the user-specified threshold

Else, set $\lambda = \lambda_i$

if level is equal to 1

- Evaluate all applicable access paths to the base relations; use add_SubPlan_errDP for error-sensitive base relations; for all other relations, propagate only the cheapest-cost access path.

else

- Construct all possible joinnodes at this level from every available and feasible pair of lower-level nodes. In doing so, use add_SubPlan_errDP to add generated candidate subplans for nodes that have at least one error-sensitive base relation; propagate only the cheapest-cost plan at that level for all other nodes.

For each of these (join)nodes j

if j is made up of an error-sensitive base relation

j.SubPlanlist = filter_and_cap(j.SubPlanlist)

$SubPlanlist$ **add_SubPlan_errDP(**$SubPlanlist\ plist$**,** $SubPlan\ p$**)**
if p would have been produced in normal DP

- Set $p.DPProduced$ to be true
- $plist = plist \cup p$
- Among all plans in $plist$ where $p.DPProduced = true$, remove the DPProduced flag on all but the cheapest plan

else

$plist = plist \cup p$

return plist

$SubPlanlist$ **filter_and_cap(**$SubPlanlist\ plist$**)**

1. Let $SubPlanlist\ result = \phi$
2. Let p_{ch} be the cheapest SubPlan in the $SubPlanlist$ such that $p_{ch}.DPProduced = true$
3. Let $cost\_upper\_bound = p_{ch}.cost * (1 + \lambda/100)$
4. Sort $plist$ in the descending order of $StabilityIndex$
5. For each $SubPlanp$ in $plist$ for this join node

   if not $p.DPProduced$ and $p.cost \leq cost\_upper\_bound$ and $p.StabilityIndex \geq 1$

   if we are at the top level of the plan tree or $\mid result \cup p \mid \leq \hat{l}_{level}$

   $result = result \cup p$
6. $result = result \cup p_{ch}$
7. return result

Figure 1.6: The InternalFixed Algorithm

distributed data) and **TPC-DS**-based query templates (with skewed data) are considered in our study. All 2-dimensional diagrams have been generated with a grid resolution of 300 and 3-dimensional diagrams with a resolution of 100.

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

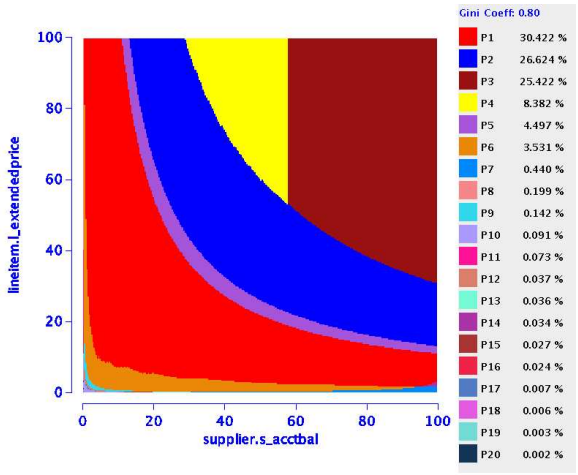
Query Distribution. The performance results shown with the suffix *exp* in this section are for plan diagrams generated with *exponentially* distributed locations for the query points across the selectivity space, resulting in higher query densities near the selectivity axes and towards the origin. All other experiments have a *uniform* distribution of query locations.

Query Template	Algo	Time (min, sec)	Pro-rated time	Plans	AvgSERF
QT5	DP	8 m 27 s	8 m 27 s	14	-
	Root	11 m 21 s	8 m 34 s	6	0.24
	InternalFixed	16 m 38 s	14 m 45 s	6	0.96
QT7	DP	7 m 2 s	7 m 3 s	8	-
	Root	9 m 11 s	7 m 6 s	7	-14.32
	InternalFixed	11 m 46 s	10 m 55 s	2	0.89
QT8	DP	9 m 44 s	9 m 44 s	20	-
	Root	14 m 30 s	9 m 54 s	9	-0.34
	InternalFixed	22 m 46 s	20 m 17 s	2	0.96
QT9	DP	9 m 11 s	9 m 11 s	6	-
	Root	10 m 25 s	9 m 20 s	4	0.81
	InternalFixed	17 m 20 s	16 m 50 s	1	0.99
QT10	DP	3 m 36 s	3 m 36 s	12	-
	Root	3 m 52 s	3 m 37 s	4	0.2
	InternalFixed	5 m 2 s	4 m 45 s	3	0.99

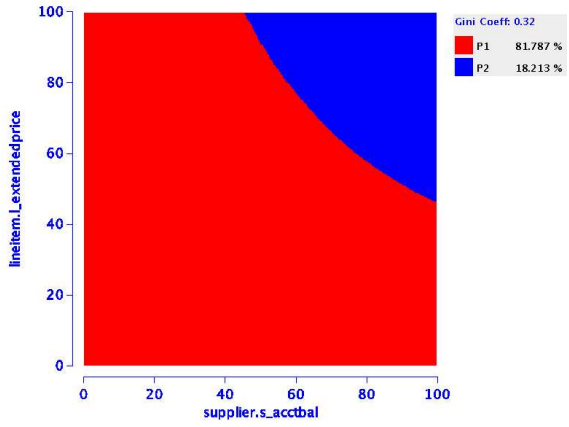
Table 1.2: Comparative performance and overheads of various strategies

Performance Metrics. In the remainder of this section, we first show some comparative examples of the computational overheads and error-resistance characteristics of the various algorithms we have proposed. Then, we evaluate our algorithm of choice and compare it to SEER-based post-facto reduction with regard to the following performance parameters: diagram reduction quality, the error-resistance obtained through reduction and computational efficiency.

InternalFixed in the experiments refers to the algorithm where the stability index is calculated using the Corner heuristic mentioned previously, and the stability criterion applied at every level of the plan tree. The IF (MaxSel) refers to a lite version of the InternalFixed algorithm where the stability is calculated only on the basis of the cost at the corner of the space with highest selectivity. We also tried another strategy where propagation of plans were done solely on the basis of the cost at the lower levels of the tree, and the stability metric was used only at the final level to decide the plan to be output. We refer to this algorithm as IF (CostOnly). For the Internal algorithms, both λ_i and λ were set to 10% and a uniform plan cap of 5 plans was applied at each node. Post-facto reduction was performed using the SEER algorithm [10] with λ set to 10%. In order to factor out the effects of the caching problem faced in PostgreSQL, we also present pro-rated running times for our algorithms where the time taken to cost all candidate sub-plans outside our λ thresholds at the remote points is deducted from the total running time. A sample plan diagram for DP shown in Figure 1.7(a) and the corresponding diagram obtained when InternalFixed is used is shown in Figure 1.7(b).



(a) **DP**



(b) **InternalFixed (λ = 10%)**

Figure 1.7: **Sample Plan Diagram for DP and InternalFixed (QT8)**

Query Template	DP Plans	InternalFixed						SEER		
		Plans	Non-POSP	Replaced (%)	MinSERF	AvgSERF	Negative (%)	Plans	MinSERF	AvgSERF
QT5	14	6	1	67.98	-5.99	0.12	0.01	5	0.78	0.96
QT7	8	2	1	59.81	0.19	0.24	0.00	2	0.35	0.89
QT8	20	2	0	72.05	-0.26	0.89	1.07	2	0.41	0.96
QT9	6	1	0	33.81	0.38	0.99	0.00	1	0.38	0.99
QT10	12	3	0	43.22	-0.40	0.99	0.07	2	0.92	0.99
Q8_3d	59	21	3	54.23	-15.63	0.13	0.12	18	0.00	0.84
Q9_3d	64	11	2	57.44	-35.21	0.25	0.88	12	0.11	0.66
QT5_exp	27	6	1	98.43	-93.51	0.11	4.43	5	0.41	0.98
QT7_exp	17	2	1	85.04	-48.01	-0.04	12.43	3	0.07	0.97
QT8_exp	31	2	0	97.84	-0.24	0.73	0.01	2	0.40	0.73
QT9_exp	9	1	1	25.55	0.61	0.99	0.00	2	0.54	0.99
QT10_exp	16	8	1	85.08	-21.15	0.91	0.75	2	0.69	0.95
Q8_3d_exp	118	33	5	64.66	-15.63	0.13	0.12	30	0.00	0.84
Q9_3d_exp	82	22	4	43.22	-35.21	0.25	0.88	12	0.11	0.66
AIQT8	38	2	1	86.07	-30.86	0.12	1.76	4	0.23	0.94
AIQT9	14	1	0	41.52	-161.48	-1.00	4.43	4	0.60	0.94
AIQT10	25	3	2	74.38	-1.76	0.89	1.20	5	0.27	0.96
AIQT8_3d	90	34	3	66.33	-34.23	0.33	1.54	33	0.00	0.44
AIQT9_3d	107	38	4	21.58	-15.23	0.21	2.99	34	0.00	0.56
AIQT8_exp	77	2	1	80.54	-80.34	0.06	2.23	4	0.23	0.93
AIQT9_exp	31	2	1	31.64	-123.44	-1.22	6.87	4	0.60	0.95
AIQT10_exp	32	4	2	68.44	-20.63	0.78	1.19	5	0.27	0.95
DS18	20	3	1	64.28	-36.03	0.89	0.02	2	0.43	0.83
DS19	36	2	0	74.38	0.21	0.76	0.00	2	0.21	0.76
DS18_exp	29	4	2	77.35	-42.63	0.84	0.20	2	0.44	0.97
DS19_exp	49	2	0	75.86	0.16	0.77	0.00	2	0.16	0.77

Table 1.3: Performance comparison of InternalFixed and SEER-based post-facto reduction

Query Template	DP	InternalFixed	Pro-rated IF	PIF / DP
QT5	8 min 27 sec	16 min 38 sec	14 min 45 sec	1.75
QT7	7 min 2 sec	11 min 46 sec	10 min 55 sec	1.55
QT8	9 min 44 sec	22 min 46 sec	20 min 17 sec	2.08
QT9	9 min 11 sec	17 min 20 sec	16 min 50 sec	1.83
QT10	3 min 36 sec	5 min 2 sec	4 min 45 sec	1.32
Q8_3d	1 hr 32 min	4 hr 1 min	3 hr 55 min	2.55
Q9_3d	1 hr 33 min	3 hr 24 min	3 hr 14 min	2.09
QT5_exp	7 min 11 sec	16 min 30 sec	14 min 40 sec	2.04
QT7_exp	6 min 39 sec	11 min 16 sec	10 min 51 sec	1.63
QT8_exp	8 min 16 sec	25 min 10 sec	22 min 11 sec	2.68
QT9_exp	8 min 4 sec	19 min 21 sec	17 min 55 sec	2.22
QT10_exp	3 min 16 sec	4 min 56 sec	4 min 40 sec	1.43
Q8_3d_exp	1 hr 27 min	4 hr 2 min	3 hr 36 min	2.48
Q9_3d_exp	1 hr 41 min	3 hr 45 min	3 hr 35 min	2.13
AIQT8	11 min 28 sec	24 min 58 sec	20 min 21 sec	1.77
AIQT9	10 min 49 sec	23 min 27 sec	22 min 28 sec	2.08
AIQT10	4 min 30 sec	7 min 8 sec	6 min 40 sec	1.48
AIQT8_3d	2 hr 5 min	6 hr 6 min	5 hr 40 min	2.72
AIQT9_3d	2 hr 2 min	7 hr 9 min	6 hr 41 min	3.29
AIQT8_exp	11 min 13 sec	26 min 43 sec	22 min 49 sec	2.03
AIQT9_exp	10 min 48 sec	23 min 15 sec	22 min 30 sec	2.08
AIQT10_exp	4 min 26 sec	7 min 11 sec	6 min 30 sec	1.47
DS18	8 min 40 sec	17 min 59 sec	16 min 0 sec	1.85
DS19	6 min 30 sec	13 min 18 sec	12 min 14 sec	1.88
DS18_exp	8 min 46 sec	18 min 13 sec	17 min 2 sec	1.94
DS19_exp	6 min 30 sec	15 min 23 sec	12 min 59 sec	2.00

Table 1.4: Computational Overheads of the InternalFixed algorithm

Table 1.2 shows that, though the Root, IF (MaxSel) and IF (CostOnly) algorithms have significantly lower overheads than the InternalFixed algorithm using the Corner Heuristic, in many cases, they are not generally reliable as demonstrated by the negative values for AvgSERF. The universal algorithm had running times at least greater than ten times that of the other algorithms and we do not present the results for it here.

From Table 1.3 we see that in most cases, the AvgSERF value is positive and increases resistance to selectivity errors. Though a good fraction of the MinSERF values are negative, we see that the fraction of values that have a negative SERF value is very small, generally less than 2%. In the AIQT9 and AIQT9_exp, the low average SERF value is caused by a few pairs of estimated and actual locations for a replacement plan we chose over the optimizer's optimal plan; on removing these negative values, the average SERF value becomes 0.36 and 0.33 respectively. Our plan cardinalities are also very low, thereby also garnering the other benefits got by post-facto reduction.

Finally, Table 1.4 shows that we are able to achieve both a good quality reduced diagram and high error-resistance with an overhead of about two times for 2-dimensional spaces and a little over three times for 3-dimensional selectivity spaces.

1.7 Related Work

Over the last decade, a variety of *compile-time* strategies have been proposed to identify robust plans. For example, in the Least Expected Cost (LEC) approach [5, 6], 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 [13] 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.

Finally, as mentioned in the Introduction, in our own recent work [10], a post-facto approach to determining robust plans was taken. Here, a "plan diagram" [26], which is a color-coded pictorial enumeration of the plan choices of the optimizer over the relational selectivity space, is first generated. On this diagram, "anorexic reduction algorithms" [9, 10] that operate with low values of λ are employed to convert it to a much simpler picture featuring significantly fewer plans. The reduction process is usually highly successful in removing locally optimal plans that suffer from globally volatile behavior, with the few retained plans typically exhibiting good error resistance characteristics.

There are several critical differences between the techniques proposed in this chapter and the post-facto approach, as outlined in detail in the Introduction, the most important of course being that we implement an online approach based on individual query instances, and not requiring any global summary information.

1.8 Conclusions and Future work

We have shown in this chapter that it is possible to reasonably efficiently incorporate stability criteria in the DP-based optimization process that is the cornerstone of modern industrial-strength database query optimizers. Specifically, we proposed the InternalFixed algorithm that strikes a balance between the competing demands of enriching the candidate space for replacement plans, and the computational overheads involved in this process. Our extensive set of experiments, which covered a variety of logical and physical designs, indicate that a significant degree of robustness can be obtained with relatively minor conceptual changes to current optimizers, especially those that already support a foreign-plan-costing feature. We hope that our promising results would encourage commercial database vendors to incorporate such stability considerations in their optimization framework.

Chapter 2

Enhancements to the Picasso Query Optimizer Visualizer tool

2.1 Introduction

Modern database systems use a query optimizer to identify the most efficient strategy to execute the SQL queries that are submitted by users. The efficiency of the strategies, called plans, is usually measured in terms of query response time. For a given database and system configuration, the optimal plan choice given by a cost-based query optimizer is mainly the function of the selectivities of the relations participating in the query. The selectivity is defined as the estimated number of tuples or rows of a relation that are relevant for producing the result of the query. A Picasso *query template* is an SQL query that additionally features predicates of the form "relation.attribute :varies". These attributes are termed as Picasso Selectivity Predicates (PSP). Each such query template defines an n-dimensional relational selectivity space, where n is the number of PSP's. The response to the variation of selectivity of each of the PSP relations over the range 0 to 100% characterizes the optimizer behavior over this selectivity space.

Picasso [25] is a database query optimizer visualizer software developed at the Database Systems Lab [29], Indian Institute of Science. Given a query template that defines a relational selectivity space and a choice of database engine, Picasso generates a variety of diagrams that characterize the behavior of the engine's optimizer over this relational selectivity space. The

diagrams include the (compilation) plan diagram, which is a color-coded pictorial enumeration of the execution plan choices; the Compilation Cost Diagram, a visualization of the associated estimated plan execution costs; the Compilation Cardinality Diagram which is a visualization of the associated estimated query result cardinalities and the Reduced plan diagram that shows the extent to which the original plan diagram may be simplified (by replacing some of the plans with their siblings in the plan diagram) without increasing the estimated cost of any individual query by more than a user-specified threshold value [28]. Picasso can also display visualizations of plan trees or highlight the differences between a selected pair of plans in the plan diagram, or display plan trees of plans two database engines produce at a given query point in the plan diagram. Picasso also has support for Abstract plan diagrams which are a visualization of the behavior of a selected plan in the plan diagram, when the optimizer is requested to use this specific plan throughout the selectivity space. This feature is currently fully operational only on the Microsoft SQL Server and Sybase database engines.

Apart from query compilation-related diagrams, Picasso also produces the execution counterparts of the cost and cardinality diagrams where the Execution Cost Diagram is based on the query response times and the Execution Cardinality Diagram represents the actual query result cardinalities over the given selectivity space.

PostgreSQL [19] is a scalable, SQL compliant, object-relational database management system. It is free software and its source code is freely available for download and modification under the BSD license. The PostgreSQL query optimizer performs a near-exhaustive search over the space of alternative strategies while deciding on the optimal plan to execute a given query using an algorithm first introduced in IBM's System R database [16]. The PostgreSQL optimizer does not support any alternate (sub-optimal) plan costing mechanisms as of its current release.

The rest of this chapter highlights the various enhancements made to Picasso and PostgreSQL. This is followed by some experimental results indicating the performance of the changed plan diagram generation time estimator. We then conclude and outline some future enhancement avenues.

2.2 Enhancements to Picasso (Server-side)

2.2.1 Improvement in estimation of time to generate compilation plan diagrams

Old plan diagram generation time Estimator

In the earlier versions of Picasso, the estimator worked as follows: The two farthest points on the principal diagonal of the selectivity space corresponding to selectivity values (0.5%, ... , 0.5%) and (99.5%, ... , 99.5%) were compiled and their times noted. The average of these were taken and extrapolated over the entire selectivity space (depending upon the number of query points). However, the performance of this estimator was observed to be poor in practice. This is because of multiple reasons. First, because it is the first time a JDBC call is made and the database engine invoked, it takes some time for the operating system to get engine process code from swap space into main memory, and thus this estimate would be much longer than that when Picasso is actually in the process of generating a plan diagram when the database engine is constantly being accessed, there being no latency of getting the engine code into memory. Even if this problem is rectified, because averaging is done, even if exactly one of the two estimates were correct, the estimate would be off because of the other wrong value. The other problem was that the timer provided in Java, though it provides values in milliseconds, the value itself is actually updated less frequently depending on the underlying operating system (nearly 10 milliseconds on both Windows and GNU/Linux platforms). Thus, if a query compiled in less than that amount of time, and the clock was checked before and after this compilation, it is possible that the value returned is the same, thus giving a value of 0 milliseconds for the compilation of this query, resulting in a totally wrong estimate.

New plan diagram generation time Estimator

The estimator code was changed to now extrapolate the median of five estimates uniformly spaced over the principal diagonal of the selectivity space. Five was chosen since it was not too high a number while still providing decent accuracy - higher numbers would have lead to

a longer time in providing estimates, and lower numbers lead to decreased accuracy. The new estimator almost always gives an estimate within five seconds, and in the worst case has been seen to provide an estimate in about ten seconds, which can be considered as an acceptably long time that a user of Picasso can wait for a plan diagram estimate. The median was taken instead of the average so that the estimate is not affected by the boundary values. To solve the precision problem of the timer, if the value given is 0 milliseconds, it is changed to 10 milliseconds. Note that this value will be used to calculate the final estimate only if a majority (here: three or more) queries compile in under 10 milliseconds.

2.2.2 Porting of Picasso to Informix Dynamic Server

IBM's Informix Dynamic Server did not have any facility to store execution plans generated by the query optimizer into tables or have any histograms in SQL accessible format. We worked with the IBM team in order to incorporate these features into Informix and then ported Picasso to read plan and histogram tables to enable the generation of plan diagrams on Informix as well. The following table structure was chosen for the plan table:

```
    CREATE TABLE SQEXPLAIN_PLAN
(
STATEMENT_ID INTEGER,
OPERATION VARCHAR(30),
OPTIONS VARCHAR(255),
OBJECT_NAME VARCHAR(30),
ID INTEGER,
PARENT_ID INTEGER,
COST DECIMAL(12,2),
CARDINALITY DECIMAL(12,2)
);
```

where *Statement_id* holds the unique ID for the given statement that is passed from *EXPLAIN PLAN FOR* or generated depending on the context, *Operation* specifies which operation is being done at that node in the plan: for example, Dynamic Hash Join, Nested loop join, Index Path or Sequential Scan, *Options* are parameters for the *Operation* specified above, *Object_name* is the name of the base table on which the operation operates, *Id* holds the ID unique for that node in the plan, *Parent_id* holds the ID of the parent node of this current node in the plan, *Cost* holds the estimated cost of computing the sub-tree below this node in the plan, and *Cardinality* holds the estimate of the cardinality of the result of the sub-tree below this node in the plan.

The following table structure was chosen for the equi-depth histogram table:

```
CREATE TABLE SQEXPLAIN_HIST
(
TABID INTEGER,
COLNO INTEGER,
BINNO INTEGER,
BINSIZE INTEGER,
FREQUENCY INTEGER,
BOUNDVAL VARCHAR(255)
);
```

where *Tabid* holds table ID comparable with systables.tabid and syscolumns.tabid, *Colno* holds column number comparable with the syscolumns.colno, *Binno* holds the bin number, *Binsize* holds the size of the bin, *Frequency* is number of distinct values in the bin, and *Boundval* contains the upper bound within the bin. *systables* and *syscolumns* are already existing Informix tables that have information about all tables and columns currently in the database respectively.

2.3 Enhancements to Picasso (Client-side)

2.3.1 Dynamic manipulation of Picasso user settings

Picasso has various user-settable settings such as low video (which is generally set on machines with low video memory to disable caching of the display so that diagrams still display correctly), the plan diagram reduction algorithm and the desired number of plans, thresholds for the Selectivity Log and cost domination, debug modes and other default values such as the server port and plan diagram reduction threshold. Earlier, these settings had to be set manually by the user in the Java source file PicassoSettings.java and then the entire code had to be recompiled. This was cumbersome and also it was not possible to change these settings in binary-only releases. So, support was added to read these settings dynamically through a settings file called local_conf when Picasso is started; the format of this file was made ASCII text so that users could directly manipulate the file if they chose to. Also, a dialog was added to the client as in Figure 2.1 using which the user could manipulate these settings. This ensures that the local settings of the user can be changed dynamically as well as that they are persistent, i. e. will remain even if the Picasso server and/or client are restarted.

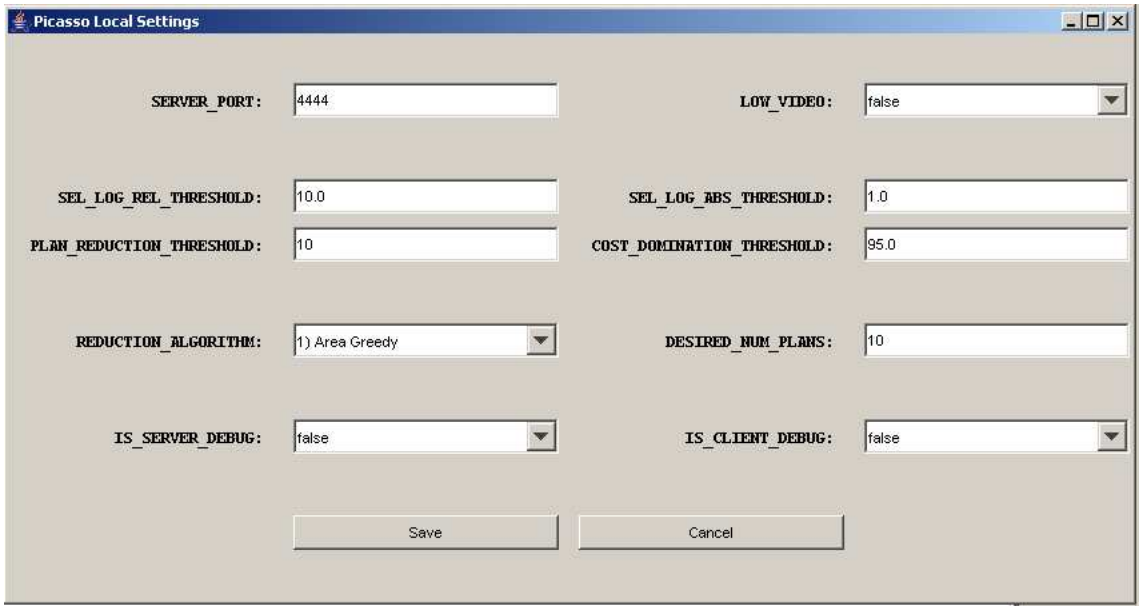


Figure 2.1: Picasso Local Settings

2.3.2 Display of slice as well as total maximum and minimum cardinality and cost values and plan counts

If the number of Picasso Selectivity Predicates is three or above, the selectivity space is a (hyper)volume. However, it is only possible to easily visualize a 2-d slice out of this. Picasso allows the user to select which slice out of the selectivity volume (s)he wants to view and displays this slice. However, earlier, the maximum and minimum cardinality and cost values as well as plan count displayed represented information about the current slice only. This made it hard to understand the behavior of the entire space as well as the relative behavior of this slice with respect to the entire volume. So, the Picasso client was augmented to show maximum and minimum cardinality and cost values as well as plan counts over both the current slice being displayed as well as the entire selectivity (hyper)volume as shown in Figure 2.2. Also, earlier, the selectivity percentages of the third and higher dimensions (to choose the required slice) had to be entered manually, and Picasso would display the nearest slice; this was cumbersome and the user needed to guess the correspondence of the selectivity value with the given slice - now a dropdown is provided as in Figure 2.3 filled with values depending upon the diagram resolution and query point distribution (uniform or exponential) and the user can select the required selectivity value, and thus the slice.

2.3.3 Saving diagram information into files

Picasso is based on a client-server model and the client and server communicate using packets. A diagram packet contains the entire description of the plan diagram including the resolution, total number of plans, maximum and minimum costs and cardinalities, selectivity values and their corresponding constants, information about the plan number, cost and cardinality of each query point compiled or executed, as well as query-specific information such as the QTD identifier and the actual query template string, the query execution type, query point distribution, number of dimensions, etc. Picasso uses three main types of high level packets. These are:

- The compilation diagram packet, which has information about the plan diagram, the compilation cost diagram and the compilation cardinality diagram.

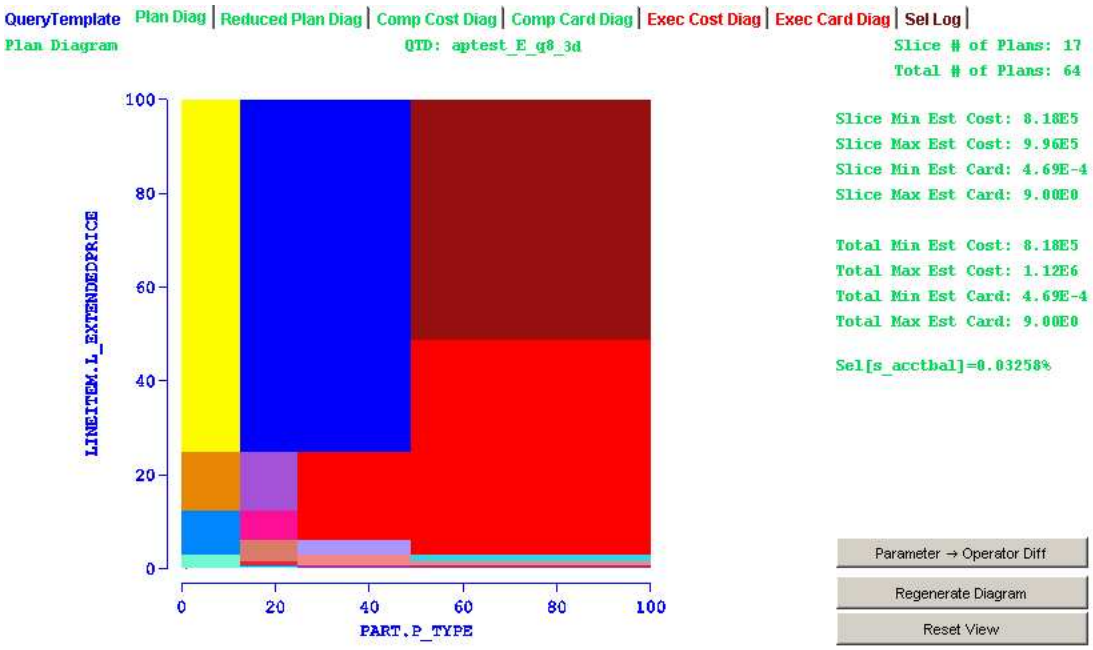


Figure 2.2: Slice and Total Cost/Cardinality/PlanCount display

- The execution diagram packet, which has information about the execution cost and cardinalities.

- The reduced plan diagram packet, which has information about plan numbering once plan diagram Reduction has been applied on the original plan diagram. This is a front-end only packet, and is handled completely at the client end.

Code was added to enable the saving of these three packets into external binary files with the extension .pkt. The benefits of saving into external files are many. One is to take a backup of the diagram information. Also, Picasso uses ViSAD (Visualization for Algorithm Development) [30] which is a Java component library for interactive and collaborative visualization and analysis of numerical data based on Java3d for visualization purposes in the client front-end. However, Java3d, though works well with Sun's java implementation, does not work with certain distributions of Java such as that provided by IBM. Thus, for reasons of universal

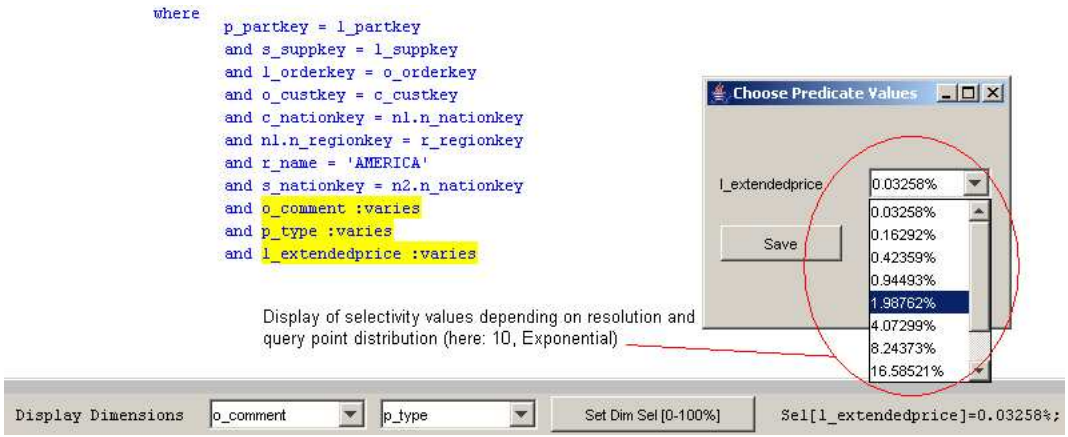


Figure 2.3: Slice selection dropdown - for 3rd (and greater) PSP's

accessibility as well as due to legal restrictions in certain companies, it becomes important to provide information collected and created by Picasso into a textual format or a format which can be read through a computer program so that new front ends may be created on platforms which cannot display this information directly through the Picasso client. So, support for saving diagram information into external files was added. The Picasso distribution was also updated with code which reads .pkt files and clearly writes all the information contained in the packet into a textual format; this would allow users to look at information contained in the .pkt files without needing a front end as well as guide front end developers as to how to programmatically access the fields in the packet.

2.3.4 Regular Expression Search for the QTD List

When Picasso generates a plan diagram, it saves the information about the diagram persistently in the database, so that it can be directly retrieved the next time the user wants to see a plan diagram over the same query template and exactly the same settings. For this purpose, a unique user specified identifier string known as the query template Descriptor (QTD) is assigned before the generation of every plan diagram. We have observed that these QTD lists can grow really long over the course of time making it hard for the user to scroll through and select the required QTD of a diagram which was generated earlier. To solve this problem, a regular expression based search facility was added to the search box which filters the QTD list

on the basis of user entry as shown in Figure 2.4, thus making it convenient to find the required QTD.

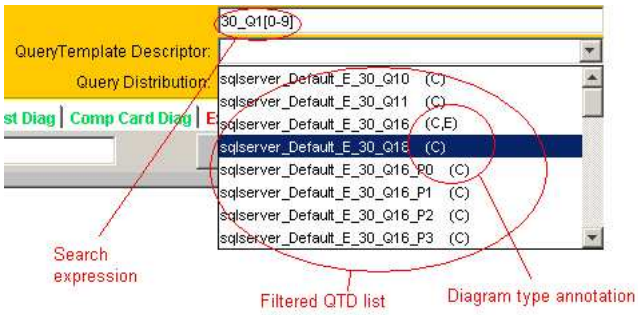


Figure 2.4: Regular Expression Search in the QTD Dropdown and QTD annotation

2.3.5 Annotation of QTD's depending upon the query execution type

Picasso can be used to generate both compile-time diagrams (got using plans generated from EXPLAIN output) as well as execution diagrams (got by actually executing queries and observing their output count, time, etc.). Generally, execution diagrams take far longer than compilation diagrams to produce, and execution diagrams are always produced after the corresponding compilation diagram is produced so that values from the compilation diagram can be used to provide an estimate of the execution diagram production time as well as to aid comparison of compilation and execution diagrams. By looking at the QTD, the user may not be able to figure out whether both the compilation and execution diagrams have been generated for it, or only the compilation diagram. Earlier, to do this, the user had to choose the compilation diagram, click on an execution tab such as the Exec Cost Diag tab, and see if it directly gave a diagram or an estimate. However, this is cumbersome and also time consuming since the execution diagram time estimate itself takes quite some time to calculate. Note that this problem cannot be got away with by aptly naming the QTD as it is possible that the user initially generates only the compilation diagram for a particular query template and settings, and later decides to generate the corresponding execution diagram as well, at which point (s)he cannot change the QTD identifier. Thus, the QTD dropdown list itself is now annotated with (C) which indicates that only a compilation diagram has been generated or (C,E) which indicates

that both the compilation and execution diagrams have been generated for that QTD. Note that this string can also be a part of the regular expression search mentioned previously; thus, if the user wanted to search for all QTD's which have their execution diagrams already generated, all (s)he had to do is to search for the string (C,E) and the list would be appropriately filtered.

2.3.6 Saving of diagram visuals into the .png format

Picasso's Save feature which can be used to save plan diagrams, cost diagrams as well as plan trees saved these diagrams into the .jpg (Joint Pictures Experts Group) format. While this format is popularly used on the internet especially for storing photographs, the compression mechanism that it uses is lossy and thus, if images such as plan diagrams are stored in the format, there is loss of quality of the image as well as smudging of text. It is important that in images of plan diagrams, the colors exactly represent the regions as they did when displayed as different colors represent distinct plans over the selectivity space. Thus, the ability to save into the popular (and thus universally readable) as well as lossless .png (Portable Network Graphics) format was added. This lossless format was chosen over others since the highly compressed .gif (Graphics Interchange Format) format did not allow for a large number of colors that were required to accurately save the plan diagram, and the .bmp (Bitmap) and .tiff (Targa Information File Format) occupied huge amounts of hard disk space per image, unlike .png files, whose size is comparable to their .jpg counterparts. Also, fonts for selectivity labels (the scales of the plan diagram) were chosen so that they appear better on the screen as well as in print (or when saved as a .png image).

2.3.7 Comparison of compilation and execution cardinalities

Current optimizers use cardinality estimates during the choosing of an appropriate plan for the given query. The correctness of these estimates severely affect the quality of the plan chosen which has direct impact on the quality of the optimizer itself. Thus, it might be important to see how correct these estimates are by comparing the estimated (compilation) cardinality values to the actual execution cardinalities over the selectivity space. While Picasso allows the

production of both compilation and execution cardinality diagrams, it is hard to visualize and compare them together - firstly, because they are in different tabs and do not display together. Even if multiple clients are used to display these diagrams side by side for comparison, it is hard to compare the values since each of these diagrams are normalized with respect to the maximum values over these individual diagrams which are generally different for the compilation and execution cardinality diagrams. An additional tab was added to Picasso which shows both the compilation and execution cardinality plots on a single 3-dimensional graph colored appropriately. The maximum and minimum compilation and execution cardinality values are also displayed as in Figure 2.5.

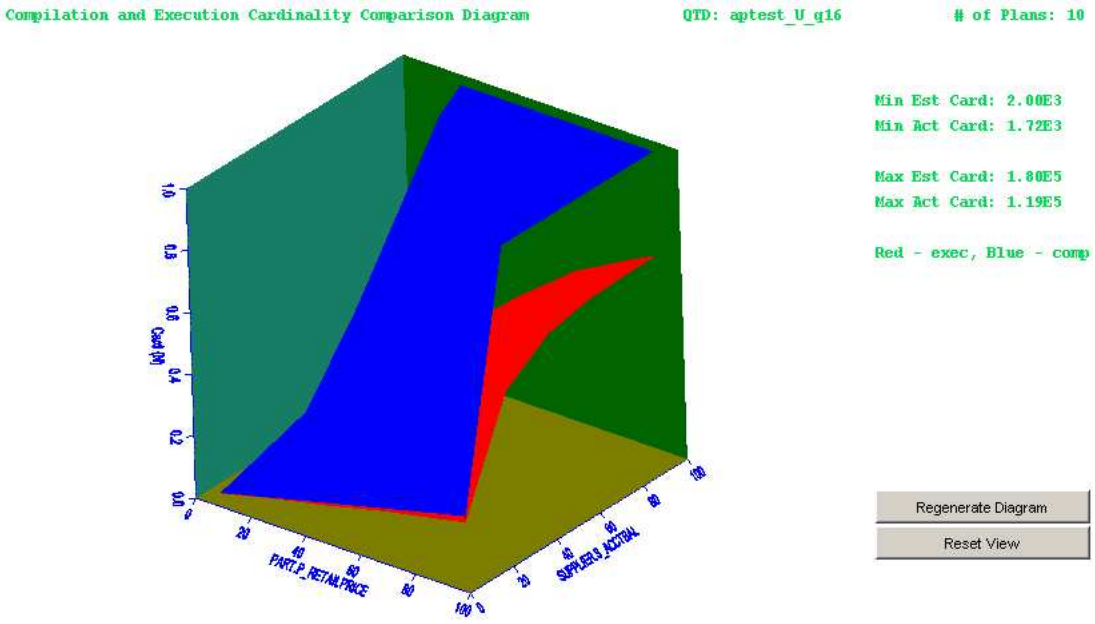


Figure 2.5: Comparison of Compilation and Execution Cardinalities

2.3.8 Client-side slicing of three and higher dimension Picasso Diagrams

In the earlier release, for diagrams which had 3 or more dimensions, Picasso required the user to specify before the diagram was read as to which 2-dimensional slice of the diagram was required to be displayed. This made the interface to read multiple slices one after the other cumbersome as the user had to repeatedly switch between the query template entry tab and the diagram display tab and also extremely slow because the client would ask the server to retrieve

the slice and the server would read the entire diagram from the database each time and then extract the required slice.

So, the server was made to send information about the entire volume rather than a slice, and slicing was performed as a client-only feature. GZIP-based compression was used to reduce the size of the diagram packet so as to reduce the one-time overhead of sending the diagram from the server to the client. Note that the entire diagram was being read even in the earlier case even when just a slice was required to be displayed in order to extract global information such as the total number of plans over the entire space as well as maximum and minimum estimated costs and cardinalities. This also made it feasible to perform plan diagram reduction over the entire diagram directly. Plan diagram reduction over the entire volume is different from reduction of each slice, since in the earlier case, neighbors from other slices can also be considered swallowers [9].

2.3.9 Compressed transmission of diagram packets with all Plan Trees

Picasso allows the user to view the plan tree structure for any plan in the plan diagram or visually view the difference between two plan trees in the diagram. Earlier, this required the client to go to the server for the plan trees, and then they were displayed (after tree differencing, if required). This made the process of getting plan trees very slow, since for each tree, the client would have to communicate with the server, and the server in turn would have to get the required tree from the database. To improve this, all plan trees were sent along with the Picasso diagram. This one time overhead of sending all trees with the diagram is mitigated due to the use of compression.

The Parameter↔Operator difference feature allows the user to display a plan diagram when two plans are considered different either only on the basis of operators in the plan tree or also with the consideration of parameters to these operators. This earlier required the client to go to the server, and the server in turn to go back to the database to get all the plan trees, and then conversion was performed on the client. Now, however, since all plan trees are directly available on the client side, this feature has been made client-only, thus improving its speed.

2.3.10 Miscellaneous enhancements

Picasso was made available as a Windows Setup executable with only binaries included so that people could use Picasso without looking at its code which is a legal restriction at many companies, and also for ease of installation. The Picasso documentation which includes the user manual was made available in the .pdf (Portable Document Format) format. The server console output earlier displayed only numeric message ID's for messages received from the client. Now, descriptive identifiers were added to the output which made debugging easier. Also, major and minor bugs in the earlier version of Picasso were fixed and other minor improvements were made.

2.4 Enhancements to PostgreSQL

2.4.1 Multiple plan output in PostgreSQL

Plans are accessed through the PostgreSQL client by executing a query of the form 'EXPLAIN ...' where the *explain* keyword is followed by the actual query whose optimal plan is needed to be obtained. However, it might sometimes be necessary to obtain sub-optimal plans over the search space too. This might be to observe cost differences of plans whose costs are near that of the optimal plan, to aid in generation of plan diagrams using sampling [7], etc. Thus, functionality was added to PostgreSQL to output multiple plans given a single *explain* query. The PostgreSQL code was also modified so as to output certain plans considered by the optimizer before making its final choice which are within a specified threshold of the optimal (and cheapest) plan generated by the optimizer.

2.4.2 Remote costing in PostgreSQL

For a given point in the selectivity space, using its cost model, the database engine generates a query execution plan which has the lowest estimated execution cost and this plan is deemed optimal for the query, and is used to execute the query. A feature which will enable users to cost the optimal execution plan at one point in the selectivity space at some other point in the space

where it may or may not be optimal has multiple uses. This would provide for the abstract-plan like diagrams on the PostgreSQL engine. In Picasso, with Microsoft SQL Server and Sybase, the Abstract Plan feature is implemented as follows. The required plan whose behavior over the entire selectivity space has to be observed is obtained by first selecting a point in the plan diagram with that plan and then acquiring from the engine in the XML format, the plan at that point. This XML string is attached to the query template and the template is now processed like in a normal plan diagram generation. Picasso would now generate queries with actual values over the selectivity space and each of these queries would be appended with this XML string, thus providing a hint to the query optimizer for each of the points over the space. However, this method is expensive (with respect to time) because the input XML string that is tagged along with the input query has to be parsed and checked for syntactic and semantic correctness because the optimizer has no way of knowing that it was itself responsible in generating that output plan. Note that this will be done at each point over the selectivity space when the Abstract Plan based diagram is being generated. Also, the Abstract Plan output provided by the optimizer is incomplete in the sense that some sub-operators (parameters to operators) among other values are not emitted; and thus will be recalculated independently for the other selectivity point by the query optimizer, which might lead to different values from that at the original query point. Also, we have noticed that the subset of plan XML strings accepted as a hint by the Microsoft SQL Server query optimizer is a strict subset of the plans that it actually emits. This means that some plans generated by it cannot be passed back as a hint to it. Also, since the plan is passed only as a hint, it is up to the optimizer to accept or reject it and sometimes it might not be possible at all to obtain and observe the behavior of a single plan over the entire selectivity space due to this, because at certain points, the optimizer may reject the hint and re-optimize from scratch, which might lead to a different plan.

Remote-costing was implemented inside the PostgreSQL database engine - all validity checks are skipped to save time as the plan would be passed around in the optimizer module itself. Also, the optimizer always uses the plan provided and we are assured of getting the required results unlike in the other engines.

DB Engine	Opt. Level	Query template	Resolution	Estimated Time	Actual Time	Absolute Error	Relative Error
SQL Server	Default	QT2	100	1 hr 6 min	1 hr 3 min	3 min	4.76%
SQL Server	Default	QT3	100	13 min 45 sec	13 min 15 sec	30 sec	3.77%
SQL Server	Default	QT5	100	1 hr 9 min	1 hr 3 min	6 min	9.52%
SQL Server	Default	QT8	100	1 hr 54 min	1 hr 46 min	8 min	7.55%
DB2	9	QT5	100	45 min 43 sec	46 min 34 sec	-51 sec	-1.83%
DB2	5	QT8	100	24 min 18 sec	26 min 48 sec	-2 min 30 sec	-9.33%
Sybase	Default	QT9	100	35 min	25 min 11 sec	9 min 49 sec	38.98%
Sybase	Default	QT16	100	27 min 1 sec	17 min 21 sec	9 min 40 sec	55.72%
PostgreSQL	Default	QT5	300	24 min 39 sec	23 min 25 sec	1 min 14 sec	5.27%
PostgreSQL	Default	QT9	100	3 min 3 sec	2 min 43 sec	20 sec	12.27%
Oracle	Default	QT8	100	11 min 12 sec	10 min 24 sec	48 sec	7.69%
Oracle	Default	QT9	100	11 min 12 sec	13 min 40 sec	-2 min 28 sec	-18.05%

Table 2.1: Accuracy of the new plan diagram generation time estimator

2.5 Experimentation

The changed (compilation) plan diagram generation time estimator using the median of five values was tried multiple times on a number of queries and was found to be more consistent and more accurate than the earlier estimator.

The new estimator was tested with a variety of query templates over all supported database engines. The results are tabulated in Table 2.1. We find that the performance of the new estimator is good and the estimate is very close to the actual generation time in most cases. Also, the estimator generally overestimates which is as required, because it is worser to take more time than presented to the user than to finish earlier. The estimator refines its estimate as queries are compiled and displays these changing estimates on the status bar by using information about the time the already compiled queries took and extrapolating this over the remaining query points. We plotted graphs of actual remaining time vs. the estimated remaining time for these query templates to see how quickly the estimator converges to a value near the actual remaining (which gives an idea as to how many queries need to be compiled before giving a highly accurate estimate of generation time). These graphs are presented below. They show that in almost all cases, the estimator presents a very good estimate within the compilation of very few queries.

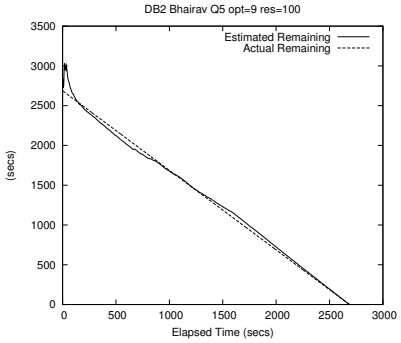


Figure 2.6: DB2, OptLevel=9, Res=100, QT5

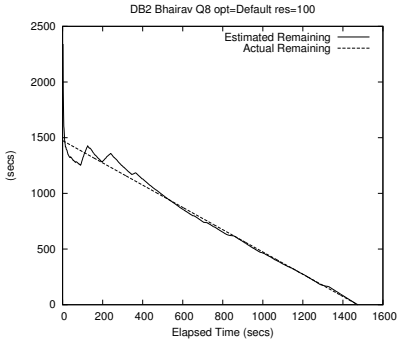


Figure 2.7: DB2, OptLevel=5, Res=100, QT8

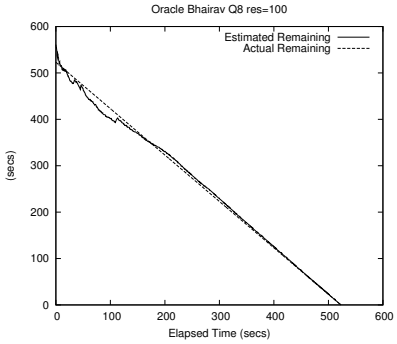


Figure 2.8: Oracle, Res=100, QT8

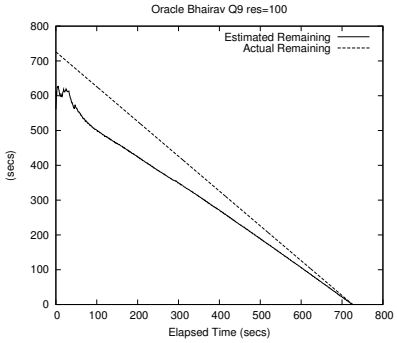


Figure 2.9: Oracle, Res=100, QT9

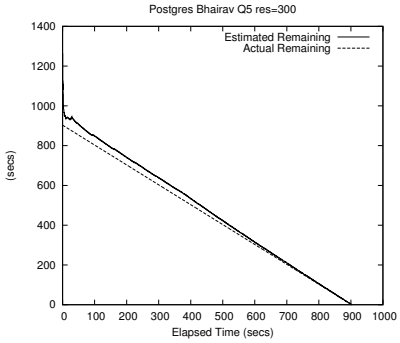


Figure 2.10: PostgreSQL, Res=300, QT5

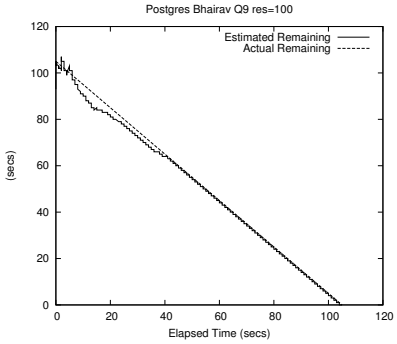


Figure 2.11: PostgreSQL, Res=100, QT9

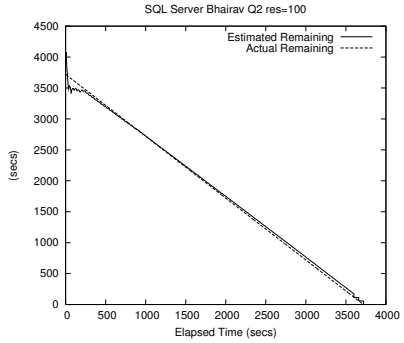


Figure 2.12: Microsoft SQL Server, Res=100, QT2

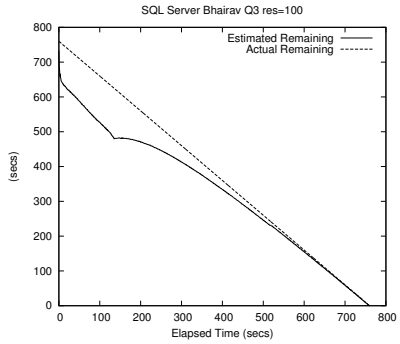


Figure 2.13: Microsoft SQL Server, Res=100, QT3

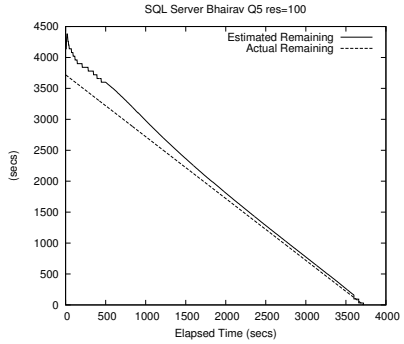


Figure 2.14: Microsoft SQL Server, Res=100, QT5

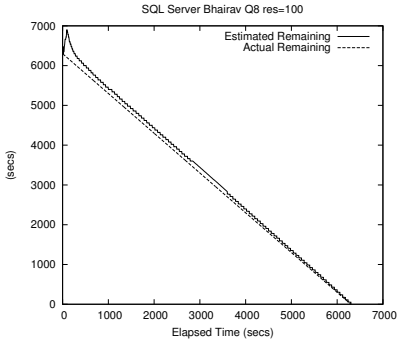


Figure 2.15: Microsoft SQL Server, Res=100, QT8

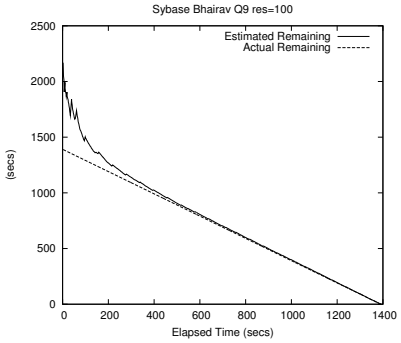


Figure 2.16: Sybase, Res=100, QT9

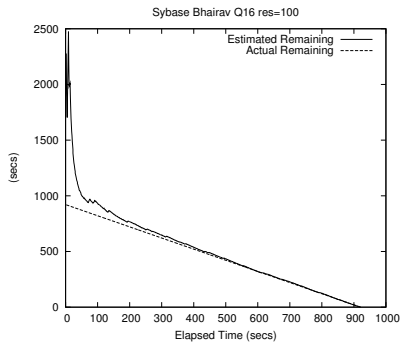


Figure 2.17: Sybase, Res=100, QT16

2.6 Conclusion and Future work

Various enhancements were made to both the client and server modules of Picasso to make it more usable and user-friendly. These included improving the compilation plan diagram generation time estimator, allowing for dynamic manipulation of Picasso user settings, saving diagram information into files, porting to IBM's Informix Dynamic Server among other improvements. Also, the commercially supported database engine PostgreSQL was augmented to output multiple (and sub-optimal) plans for a given explain query. Remote costing was also implemented on PostgreSQL.

Currently, in Picasso, diagrams can be generated only over the entire selectivity range and with an equal grid resolution on each dimension. However, for many applications, it is required to generate high-resolution diagrams over only a small part of the selectivity range. Generating such diagrams over the entire range and then filtering is a prohibitively expensive consideration. Also, sometimes users may be interested in varying the selectivity of one relation more finely than another, and that requires allowing for different resolutions for different dimensions. Allowing flexibility in these aspects would improve Picasso considerably.

In PostgreSQL, remote costing has currently been implemented internally. In order for it to be universally used, an external API must be written for it that allows the user to input a plan in some format along with the query and selectivity constant values of a remote point and in return receive the sent plan with costs that the given plan would have at the given remote point.

In Picasso, plan diagram reduction is generally done using a cost-bounding approach [26]. This principle is conservative in that it does not capture all swallowing possibilities, because it restricts its search only to the first quadrant. The benefits of doing actual costing have been observed with Microsoft's SQL Server and Sybase database engines using the Abstract Plan feature where most plan diagrams reduced to just one or a couple of plans, instead of just the low absolute number got on doing plan diagram reduction using the cost-bounding approach. Whether the same benefits can be garnered by using remote costing on PostgreSQL remains to be seen.

Bibliography

[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] 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.

[6] 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.

[7] A. Dey, S. Bhaumik, Harish D. and J. Haritsa, "Efficient Generation of Approximate Plan Diagrams", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2008.

[8] R. Guravannavar and S. Sudarshan, "Reducing Order Enforcement Cost in Complex Query Plans", *Proc. of Intl. Conf. on Data Engineering (ICDE)*, April 2007.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] *http://www.postgresql.org/docs/8.2/static/release-8-2-5.html*

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

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

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

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

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

[26] 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. *http://dsl.serc.iisc.ernet.in/publications/conference/picasso-revised.pdf*

[27] Tarun Ramsinghani, "Picasso 1.0: Design and Analysis", *Master's Thesis, CSA, IISc*, July 2007. *http://dsl.serc.iisc.ernet.in/publications/thesis/tarun.pdf*

[28] Harish D., P. Darera and J. Haritsa, "Reduction of Query Optimizer Plan Diagrams", *Tech. Rep. TR-2007-01, DSL/SERC, Indian Inst. of Science*, 2007. *http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2007-01.pdf*

[29] *http://dsl.serc.iisc.ernet.in/*

[30] *http://www.sourceforge.net/projects/visad/*