Efficient Generation of Approximate Plan Diagrams

Atreyee Dey Sourjya Bhaumik Harish D. Jayant R. Haritsa

Technical Report
TR-2008-01

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

Abstract

Given a parametrized n-dimensional SQL query template and a choice of query optimizer, a plan diagram is a color-coded pictorial enumeration of the execution plan choices of the optimizer over the query parameter space. These diagrams have proved to be a powerful metaphor for the analysis and redesign of modern optimizers, and are gaining currency in diverse industrial and academic institutions. However, their utility is adversely impacted by the impractically large computational overheads incurred when standard brute-force exhaustive approaches are used for producing high-dimension and high-resolution diagrams.

In this paper, we investigate strategies for efficiently producing high-quality approximate plan diagrams that have low plan-identity and plan-location errors. Our techniques are customized to the features available in the optimizer's API, ranging from the generic optimizers that provide only the optimal plan for a query to those which also support costing of sub-optimal plans and enumerating rank-ordered lists of plans. The techniques collectively feature both random and grid sampling, as well as interpolation techniques based on kNN classifiers , parametric query optimization and plan cost monotonicity.

Extensive experimentation with a representative set of TPC-H and TPC-DS-based query templates on industrial-strength optimizers indicates that our techniques are capable of meeting identity and location error bounds as low as 10% while incurring less than 15% of the computational overheads of the exhaustive approach. In fact, for full-featured optimizers, we can guarantee zero error with overheads of less than 10%. These approximation techniques have been implemented in the publicly available Picasso optimizer visualization tool.

1 Introduction

For a given database and system configuration, a query optimizer's execution plan choices are primarily a function of the *selectivities* of the base relations in the query. In a recent paper [16], we introduced the concept of a "plan diagram" to denote a color-coded pictorial enumeration of the plan choices of the optimizer for a parameterized query template over the relational selectivity space. For example, consider QT8, the parameterized 2D query template shown in Figure 1, based on Query 8 of TPC-H. Here, selectivity variations on the SUPPLIER and LINEITEM relations are specified through the s_acctbal :varies and l_ extendedprice :varies predicates, respectively. The associated plan diagram for QT8 is shown in Figure 2(a), produced with the Picasso optimizer visualization tool [19] on a popular commercial database engine.

In this picture [1], each colored region represents a specific plan, and a set of 89 different optimal plans, P1 through P89, cover the selectivity space. The value associated with each plan in the legend indicates the percentage area covered by that plan in the diagram – the biggest, P1, for example, covers about 22% of the space, whereas the smallest, P89, is chosen in only 0.001% of the space.

Applications of Plan Diagrams

Since their introduction in 2005 [16], plan diagrams have proved to be a powerful metaphor for the analysis and redesign of industrial-strength database query optimizers. For example, as evident from

[1] The figures in this paper should ideally be viewed from a color copy, as the gray-scale version may not clearly register the features.

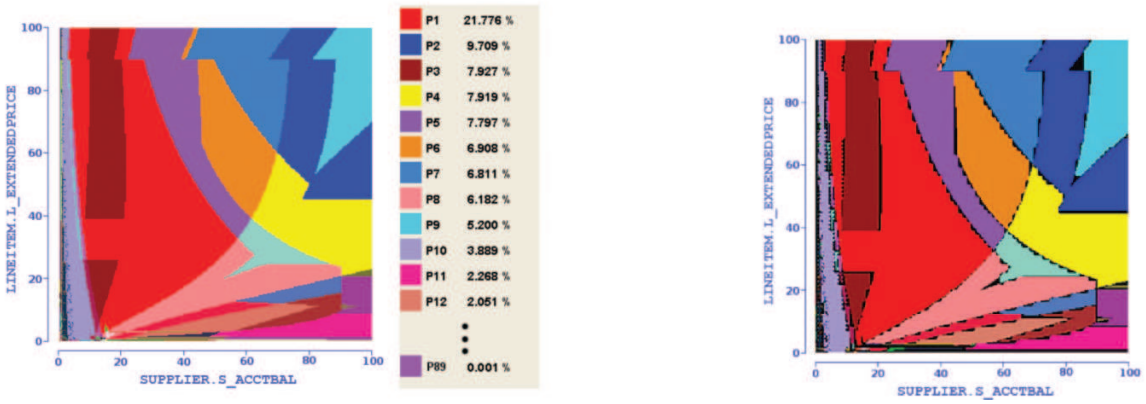
```
select o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume)
from (select YEAR(o_orderdate) as o_year, l_extendedprice * (1 - l_discount) as volume, n2.n_name as
      nation
    from part, supplier, lineitem, orders, customer,
         nation n1, nation n2, region
    where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

Figure 1: **Example Query Template: QT8**

Figure 2(a), they can be surprisingly complex and dense, with a large number of plans covering the space – several such instances spanning a representative set of benchmark-based query templates on current optimizers are available at [19]. Through our extensive interactions with industrial developers, we have found that these diagrams have proved to be quite contrary to the prevailing conventional wisdom. While developers had certainly been extensively analyzing optimizer behavior on *individual queries*, plan diagrams provide a completely different perspective of behavior *over an entire space*, vividly capturing plan transition boundaries and optimality geometries. So, in a literal sense, they deliver the "big picture".

Plan diagrams are currently being used in various industrial and academic sites for a diverse set of applications including analysis of existing optimizer designs; visually carrying out optimizer regression testing; debugging new query processing features; comparing the behavior between successive optimizer versions; investigating the structural differences between neighboring plans in the space; investigating the variations in the plan choices made by competing optimizers; etc. Visual examples of non-monotonic cost behavior in commercial optimizers, indicative of modeling errors, were highlighted in [16].

A particularly compelling immediate utility of plan diagrams is that they provide the input to "plan diagram reduction" algorithms. Specifically, given a plan diagram and a cost-increase-threshold (λ) specified by the user, these reduction algorithms recolor the dense diagram to a simpler picture that features only a subset of the original plans while ensuring that the cost of no individual query point goes up by more than λ. That is, some of the original plans are "completely swallowed" by their siblings, leading to a reduced plan cardinality. It has been shown last year [9] that if users were willing to tolerate a minor cost increase of λ =20% for any query point in the diagram, relative to its original cost, the absolute number of plans in the final reduced picture could be brought down to *within or around ten*. In short, that complex plan diagrams can be made "anorexic" while retaining acceptable query processing performance. For example, the reduced version of the QT8 plan diagram (Figure 2(a))



(a) **Plan Diagram** (b) **Approximate Diagram (10% Error Bound)**

Figure 2: **Sample Plan Diagram and Approximate Plan Diagram (QT8)**

retains only 5 of the original 89 plans.

Anorexic plan diagram reduction has significant practical benefits [9], including quantifying the redundancy in the plan search space, enhancing the applicability of parametric query optimization (PQO) techniques [11, 12], identifying error-resistant and least-expected-cost plans [3, 4], and minimizing the overhead of multi-plan approaches [1, 13]. A detailed study of its application to identifying robust plans that are resistant to errors in relational selectivity estimates is available in [10].

Generation of Plan Diagrams

The generation and analysis of plan diagrams has been facilitated by our development of the Picasso optimizer visualization tool [19]. Given a multi-dimensional SQL query template like QT8 and a choice of database engine, the Picasso tool automatically produces the associated plan diagram. It is operational on several major platforms including IBM DB2, Oracle, Microsoft SQL Server, Sybase ASE and PostgreSQL. The tool, which is freely downloadable, is now in use by the development groups of several major database vendors, as also by leading industrial and academic research labs worldwide.

The diagram production strategy used in Picasso is the following: Given a d-dimensional query template and a plot resolution of r, the Picasso tool generates r^d queries that are either uniformly or exponentially (user's choice) distributed over the selectivity space. Then, for each of these query points, based on the associated selectivity values, a query with the appropriate constants instantiated is submitted to the query optimizer to be "explained" – that is, to have its optimal plan computed. After the plans corresponding to all the points are obtained, a different color is associated with each unique plan, and all query points are colored with their associated plan colors. Then, the rest of the diagram is colored by painting the region around each point with the color corresponding to its plan. For example, in a 2D plan diagram with a uniform grid resolution of 10, there are 100 real query points, and around each such point a square of dimension 10x10 is painted with the point's associated plan color.

The above exhaustive approach is eminently acceptable for diagrams with few dimensions (upto 2D)

and low resolutions (upto 100). However, it becomes impractically expensive for higher dimensions and resolutions due to the exponential growth in overheads. For example, a 3D plan diagram with a resolution of 100 on each selectivity dimension, requires invoking the optimizer 100^3 times – that is, a *million* optimizations have to be carried out. Even with a conservative estimate of about half-second per optimization, the total time required to produce the picture is close to a week! Therefore, although plan diagrams have proved to be extremely useful, their high-dimensional and high-resolution versions pose serious computational challenges.

Approximate Plan Diagrams

In this paper, we address this issue and investigate whether it is possible to efficiently produce *high-quality approximations* to plan diagrams. Denoting the true plan diagram as P and the approximation as A, there are two categories of errors that arise in this process:

Plan Identity Error (ϵ_I): This error metric refers to the possibility of the approximation missing out on a subset of the plans present in the true plan diagram. It is computed as the percentage of plans lost in A relative to P.

The ϵ_I error is challenging to control since a majority of the plans in the plan diagrams, as seen in Figure 2(a), are very small in area, and therefore hard to find.

Plan Location Error (ϵ_L): This error metric refers to the possibility of incorrectly assigning plans to query points in the approximate plan diagram. It is computed as the percentage of incorrectly (relative to P) assigned points in A.

The ϵ_L error is also challenging to control since the plan boundaries, as seen in Figure 2(a), can be highly non-linear, and are sometimes even irregular in shape [19].

Optimizer Classes

Our study shows that the ability to reduce overheads is a function of the plan-related functionalities offered by the optimizer's API, based on which we define the following three categories of optimizers:

Class I: OP Optimizers This class refers to the generic cost-based optimizers that are routinely found in virtually every enterprise database product, where the API only provides the optimal plan (OP), as determined by the optimizer, for a query.

Class II: OP + FPC Optimizers This class of optimizers additionally provide a "foreign plan costing" (FPC) feature in their API, that is, of costing plans *outside* their native optimality regions. Specifically, the feature supports the "what-if" question: "What is the estimated cost of sub-optimal plan *p* if utilized at query location *q*?". FPC has become available in the current versions of several industrial-strength optimizers, including DB2 [20] (Optimization Profile), SQL Server [21] (XML Plan), and Sybase [22] (Abstract Plan).

Class III: OP + FPC + PRL Optimizers This class of optimizers support, in addition to FPC, an API that provides not just the best plan but a "plan-rank-list" (PRL), enumerating the top k plans

for the query. For example, with $k = 2$, both the best plan and the second-best plan are obtained when the optimizer is invoked on a query. Note that the PRL feature is *trivially implementable* in current optimizers since this list is constructed by default in the root node of the Dynamic Programming-based query optimization exercise. However, to our knowledge, it is not yet available in any of the current systems. Therefore, we showcase its utility through our own implementation in a public-domain optimizer.

Approximation Techniques and Results

For Class I (OP) and Class II (OP+FPC) optimizers, the techniques that we propose are based on a combination of sampling and interpolation, while for Class III optimizers (OP+FPC+PRL), it is purely based on interpolation. The sampling techniques include both classical random sampling and grid-based sampling, while the interpolation approaches rely on kNN-classifiers [18] , parametric query optimization (PQO) [11, 12] and plan cost monotonicity [9]. For some of the techniques, theoretical results that help to provide guaranteed bounds on the errors are available, whereas for the others, empirical evaluation is the only recourse.

We have quantitatively assessed the efficacy of the various strategies, with regard to plan identity and location errors. This has been done through extensive experimentation with a representative suite of multi-dimensional TPC-H and TPC-DS-based query templates on leading commercial and public-domain optimizers. Our results are very promising since they indicate that high-quality approximations can indeed be obtained *cheaply and consistently*, as described below.

10 percent Error Bound. Consider the case where the user expects less than *10 percent* plan identity and plan location errors. For Class I (OP) optimizers, it is possible to regularly achieve this target with *only around 15% overheads* of the brute-force exhaustive method. To put this in perspective, the earlier-mentioned one-week plan diagram can be produced in a few hours. A sample approximate diagram (having 10% identity and 10% location error) is shown in Figure 2(b), with all the erroneous locations marked in black – as can be seen, the approximation is materially faithful to the features of the true plan diagram, with the errors thinly spread across the picture and largely confined to the plan transition boundaries.

For Class II (OP+FPC) systems, it is possible to achieve a similar error performance with *only around 10%* overheads. An important point to note here is that plan costing is considerably cheaper than searching for the optimal plan. Finally, for Class III (OP+FPC+PRL) systems, the overheads come down to *less than 5%*.

1 percent Error Bound. We have also investigated the scenario where the user has the extremely stringent expectation of less than *1 percent* plan identity and location errors. For this situation, Class I and II both take around *40% overheads*, while Class III incurs only *10% overheads*.

Contributions

In a nutshell, we present in this paper a range of techniques, customized to the optimizer's API richness, for efficiently generating high-quality approximate plan diagrams. These results are summarized in Table 1, where the typical range of overheads (relative to the exhaustive approach) is shown as a function of the user's error constraint for each optimizer class.

Optimizer Class	Overheads Range (ϵ = 10%)	Overheads Range (ϵ = 1%)
Class I (OP)	10% - 15%	30% - 40%
Class II (OP + FPC)	5% - 10%	30% - 40%
Class III (OP + FPC + PRL)	$\leq$ 5%	$\leq$ 10%

Table 1: Summary Results

While not mentioned earlier, for Class II and Class III optimizers, our techniques can also produce the "cost diagram" and "cardinality diagram" associated with the plan diagram [19]. The cost diagram is a visualization of the estimated plan execution costs over the relational selectivity space, while the cardinality diagram is a similar visualization of the estimated result cardinalities.

Organization

The remainder of this paper is organized as follows: The approximation algorithms are presented in Section 2. Our experimental framework and performance results are highlighted in Section 3. Finally, in Section 4, we summarize our conclusions and outline future research avenues.

2 Approximation Algorithms

In this section, we describe our suite of strategies for the efficient generation of approximate plan diagrams. We begin with algorithms for Class I optimizers, and then describe how these techniques can be improved for Class II optimizers leveraging their foreign-plan-costing (FPC) feature. We conclude with two variants of an algorithm for Class III optimizers with FPC and plan-ranking-list (PRL) functionalities – the first version *guarantees zero error*, while the second trades error for further reduction in computational overheads.

For ease of presentation, we will assume in the following discussion that the query template is 2-dimensional – the extension to n-dimensions is straightforward and mentioned in the 5. The true plan diagram is denoted by P and the approximation as A, with the total number of query points in the diagrams denoted by m. Each query point is denoted by $q(x, y)$, corresponding to a unique query with selectivities x, y in the X and Y dimensions, respectively. The terms $p_P(q)$ and $p_A(q)$ are used to refer to the plans assigned to query point q in the P and A plan diagrams, respectively (when the context is clear, we drop the diagram subscript).

Finally, the plan identity and location errors are defined as

$$\epsilon_I = \frac{|P| - |A|}{|P|} * 100 \tag{1}$$

and

$$\epsilon_L = \frac{|p_A(q) \neq p_P(q)|}{m} * 100 \tag{2}$$

respectively.

2.1 Class I Optimizers

The approximation procedures for this class of optimizers operate in two phases:

Optimization Phase: In this phase, a subset of the query points in the plan diagram are optimized to obtain the optimal plan choices at those points.

Interpolation Phase: In this phase, the plan choices for a subset of the unoptimized points are *inferred* using the results from the Optimization Phase.

For the random sampling-based algorithms, the above two phases are sequential, whereas for the grid-sampling-based algorithms, the phases are interleaved.

2.1.1 Random Sampling with kNN Interpolation (RS_kNN)

In the RS_kNN algorithm, we first use the classical random sampling (without replacement) technique to sample query points from the plan diagram that are to be optimized during the optimization phase. Since we have empirically found that with this technique, the plan-identity error ϵ_I is almost always greater than the plan-locality error ϵ_L, the stopping criterion for the sampling is based on the former metric. The problem of finding the distinct plans in the plan diagram can be related to the classical statistical problem of finding distinct classes in a population [7]. Applying the recent results of [2], we obtain the following: Let s samples be taken on the plan diagram, let d_s be the number of distinct plans in these samples, and let f_1 denote the number of plans occurring only *once* in the samples. Then, it is highly likely that the the number of distinct plans, d, in the entire plan diagram is in the cardinality range $[d_s, d_{max}]$, where

$$d_{max} = (\frac{m}{s} - 1)f_1 + d_s \tag{3}$$

From [2], we can also deduce $\hat{d}_{ML}$, the most-likely-value estimator for d, to be

$$\hat{d}_{ML} = (\sqrt{\frac{m}{s}} - 1)f_1 + d_s \tag{4}$$

which has an expected ratio error bound of $O(\sqrt{\frac{m}{s}})$.

If we ensure that the sampling is iteratively continued until d_s is within ϵ_I of d_{max}, then it is highly likely that the number of plans found thus far in the sample is within ϵ_I of d. Therefore, the RS_kNN algorithm continuously evaluates Equation 3 to determine when the sampling process can be terminated.

Resolution / Dimension	Query Template	No. of Plans	d_{max}		d_{ML}		$HYBRID$	
			Sample %	Identity Error	Sample %	Identity Error	Sample %	Identity Error
100x100x100	QT8	190	45%	6%	20%	12%	25%	11%
100x100x100	QT9	404	47%	7%	30%	11%	35%	8%
1000 x 1000	QT8	132	55%	4%	25%	7.75%	25%	7.75%
1000 x 1000	QT21	58	15%	2%	1%	20%	10%	10.34%

Table 2: Comparative study of performance of different estimator

Our experience, as borne out by the experimental results presented in Table 2, has been that the above stopping condition may be too conservative in that it takes many more samples than is strictly

RS_kNN (QueryTemplate QT, ErrorBound ϵ, InitSamples s_0)

1. stage = 1; $s = s_0$;
2. Optimize s samples chosen uniformly at random.
3. Compute the values of d_{max} and $\hat{d}_{ML}$
4. if (stage = 1) then
5. if $d_s \geq ((1-\delta)d_{max})$ then stage = 2;
6. if (stage = 2) then
7. if $d_s \geq ((1-\epsilon)\hat{d}_{ML})$ then **stop**.
8. $s = s + s_0$
9. Go to Step 2.
10. End Algorithm RS_kNN

Figure 3: The RS_kNN Algorithm

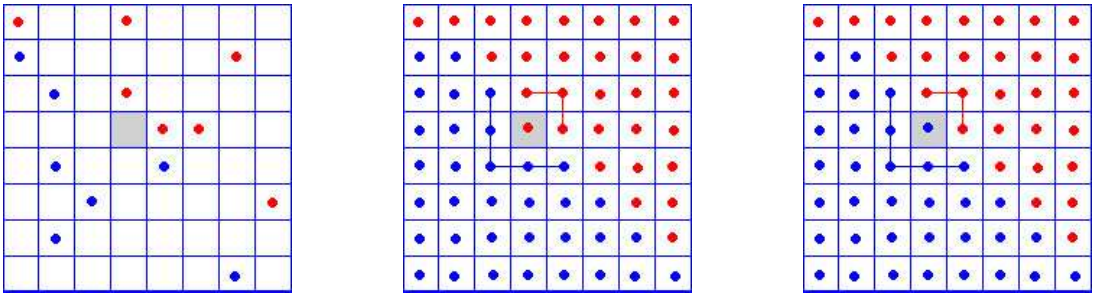
necessary. Therefore, we refine it to the following two-step process: After d_s increases to a value within a $(1-\delta)$ factor of d_{max}, then continue the sampling until d_s reaches to within a $(1-\epsilon)$ factor of $\hat{d}_{ML}$. The value of δ conducive to good performance results has been empirically determined to be 0.3. The intuition behind this method is that once the gap between d_s and d_{max} has narrowed to a sufficiently small range, then the estimator can be used as a reliable indicator of the plan cardinality in the diagram. The complete RS_kNN algorithm is shown in Figure 3. In our implementation, the initial number of samples s_0 is set to 1% of the space, and the increment in the number of samples after each iteration is also set to this value.

Interpolation. After the completion of the sampling stage, the plan choices at the non-optimized points of the plan diagram need to be inferred from the plan choices made at the sampled points. This is done using a k-Nearest Neighbor (kNN) style classification scheme [18]. Specifically, for each non-optimized point q_n, we search for the nearest (as per a distance measure) optimized point q_o, and assign the plan of q_o to q_n. If there are multiple nearest optimized points, then the plan that occurs most often in this set is chosen, and in case of a tie on this metric, a random selection is made from the contenders.

The distance between two query points $q_1(x_1, y_1)$ and $q_2(x_2, y_2)$ can be calculated using various distance metrics. We have evaluated the following three popular metrics:

- Manhattan (L_1Norm)) : $dist_{12} = abs(x_1 - x_2) + abs(y_1 - y_2)$
- Euclidean (L_2Norm) : $dist_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- Chessboard ($L_\infty Norm$) : $dist_{12} = max(abs(x_1 - x_2), abs(y_1 - y_2))$

Our experience has been that the Chessboard Distance is most suitable, since the transition boundaries between plans often tend to be aligned along the (horizontal and vertical) axes. The same metric was



(a) (b) (c)

Figure 4: **The Low Pass Filter**

also used for establishing the geometries of plan clusters in the PLASTIC optimizer value-addition tool [5, 17].

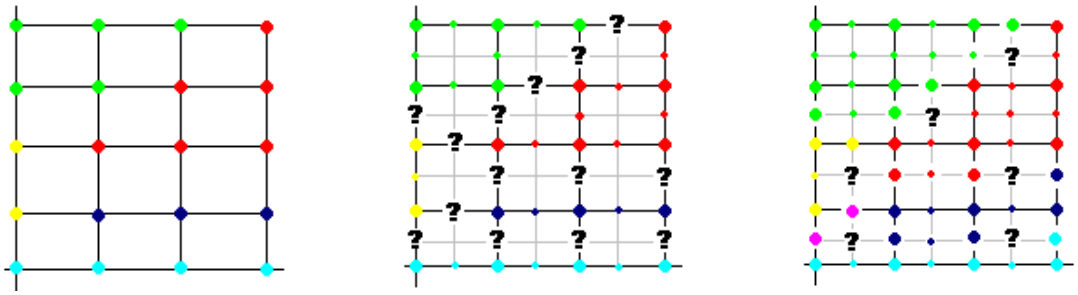
Low Pass Filter. Interpolation using the kNN scheme is well-known to result in errors along the plan boundaries [18]. To reduce the impact of this problem, we apply a low-pass filter [6] after the initial interpolation has assigned plans to all the points in the diagram. The filter operates as follows: For each non-optimized point q_n, examine all its neighbors (both optimized and non-optimized) at a distance of one to find the plan that is assigned to the majority of its neighbors. If such a plan exists, assign that plan to q_n, otherwise retain the original assignment.

The functionality of low-pass filter is illustrated in Figure 4(a) - 4(c). The sample points in a 8 X 8 square area of a plan diagram is shown in Figure 4(a). Clearly the sampling has revealed only 2 plans - Red and Blue in this area and the interpolation will determine the border between these 2 plans. Consider the point inside the greyed area. As we are searching for nearest neighbors in successive chess-board distances, we will find 2 red neighbors and 1 blue neighbor within distance 1 and this point will be classified as Red. Now consider the scenario after interpolation (Figure 4(b)). It turns out that the neighborhood of the point which was dominated by Red samples before interpolation, mostly consists of Blue points (5 out of 8 neighbors) after interpolation. It is highly likely that nearest neighbor classifier has miss-classified this point and hence we decide to assign Blue to this point by applying the low-pass filter technique (Figure 4(c)).

2.1.2 Grid Sampling with PQO Interpolation (GS_PQO)

We now turn our attention to an alternative approach based on grid sampling. Here, a low resolution grid of the plan diagram is first formed, which partitions the selectivity space into a set of smaller rectangles. The query points corresponding to the corners of all these rectangles are optimized first. Subsequently, these points are used as the seeds to determine which of the other points in the diagram are to be optimized.

Specifically, if the plans assigned to the two corners of an edge of a rectangle are the same, then the mid-point along that edge is also assigned the same plan. This is essentially a specific interpolation based on the guiding principle of the Parametric Query Optimization (PQO) literature (e.g. [11]): “If a



(a) (b) (c)

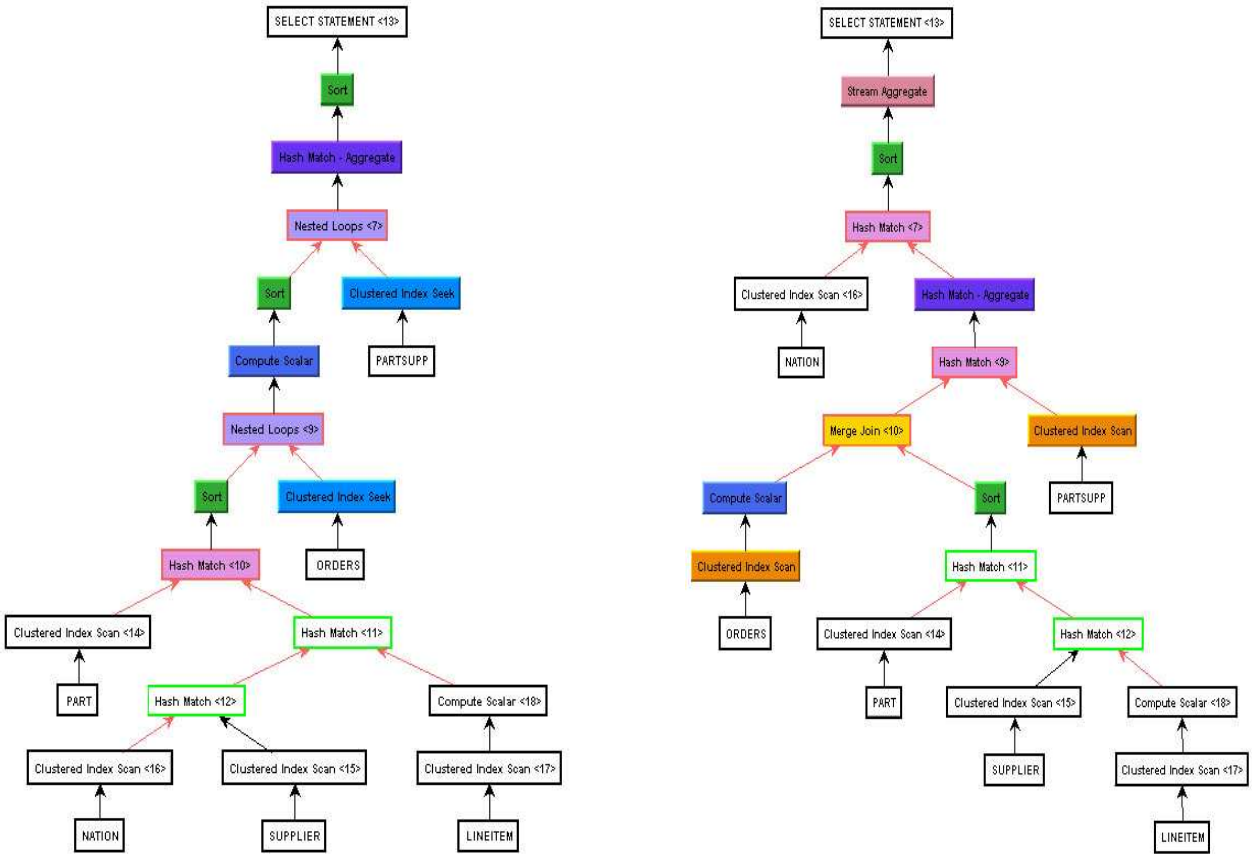
Figure 5: **The GS_PQO Algorithm**

pair of points in the selectivity space have the same optimal plan p_i, then all points along the straight line joining these two points will also have p_i as their optimal plan." At first glance, our usage of the PQO principle here may seem at odds with our earlier observation in [16] that, for industrial-strength optimizers, this principle is observed more in the breach than in the observance. However, the difference is that we are applying PQO at a "micro-level", that is, within the confines of a small rectangle in the selectivity space, whereas earlier work has effectively considered PQO as a universal truth that holds across the entire space. Our experimental experience has been that micro-PQO generally holds in all the plan diagrams that we have analyzed.

When the plans assigned to the end points of an edge are different, then the midpoint of this edge is optimized. Once the sides of a given rectangle are processed, the center-point is then processed by considering the plans lying along the "cross-hair" lines connecting the center-point to the mid-points of the four sides of the rectangle. If the two end-points on one of the cross-hairs match, then the center-point is assigned the same plan (if both cross-hairs have matching end-points, then one of the plans is chosen randomly). Now, using the cross-hairs, the rectangle is divided into four smaller rectangles, and the process recursively continues, until all points in the plan diagram have been assigned plans.

The progress of the GS_PQO algorithm is illustrated in Figure 5. In this set of pictures, each large dot indicates an optimized point, whereas each small dot indicates an inferred point. Figure 5(a) shows the state after the initial grid sampling is completed. Then, the '?' symbols in Figure 5(b) denote the set of points that are to be optimized in the following iteration as we process the sides of the rectangles. Finally, Figure 5(c) enumerates the set of points that are to be optimized while processing the cross-hairs.

We have found that a limitation of the GS_PQO algorithm is that it may perform a substantial number of unnecessary optimizations, especially when a rectangle with different plans at its endpoints features only a small number of new plans within its enclosed region. This is because GS_PQO does not distinguish between sparse and dense low-resolution rectangles. For example, if two similar-sized rectangles each have two plans featured at their four corner points, then they are divided similarly irrespective of the expected number of new plans present in the interior. We attempt to address this issue by refining the algorithm in the following manner: Assign each rectangle R with a "plan-richness" indicator $\rho(R)$ that serves to characterize the expected plan density in R, and then preferentially assign optimizations to the rectangles with higher ρ.



(a) Plan Tree T_i (b) Plan Tree T_j

Figure 6: Example of Plan Tree Difference

Our strategy to assign ρ values is as follows: Instead of merely having a *boolean* comparison at the corners of the rectangle as to whether the plans at these points are identical or not, we now dig deeper and compare the *plan operator trees* associated with these plans in order to estimate plan density. As an extreme example, consider the case where there is a left-deep tree at one corner of the rectangle, and a right-deep tree at another corner. In this situation, it seems reasonable to expect that there will be a significant number of plans in the interior of the rectangle since the process of shifting from a left-deep to a right-deep tree usually occurs in incremental intermediate steps, each corresponding to a new plan, rather than all at once – we have confirmed this observation through detailed analysis of the plan diagrams of industrial optimizers.

Plan Tree Differencing. Let the operator trees corresponding to a pair of plans p_i and p_j be denoted by T_i and T_j, respectively. Our comparison strategy is based on identifying and mapping similar operator nodes in the two trees. We use color codes to depict matching and distinct nodes of the two trees. In our description, the term *branch* is used to refer to any connected chain of unary nodes between a pair of binary nodes, or between a binary node and a leaf, in these trees. Branches are directed from the

lower node to the higher node. The matching proceeds as follows:

1. First, all the leaf nodes (relations) and all the nodes with binary inputs (typically join nodes) are identified for T_i and T_j.

2. A leaf of T_i is matched with a leaf of T_j if and only if they both have the same relation name. In the situation that there are multiple matches available (that is, if the same relation name appears in multiple leaves), an edit-distance computation is made between the branches of all pairs of matching leaves between T_i and T_j. The assignments are then made in increasing order of edit-distances.

3. A binary node of T_i is matched with a binary node of T_j if the set of base relations that are processed is the same. If the node operator names and the left and right inputs are identical (in terms of base relations), the nodes are made white. However, if the node operator names are different, or if the left and right input relation subsets are different, then the nodes are colored.

4. A minimal edit-distance computation is made between the branches arising out of each pair of matched nodes, and the nodes that have to be added or deleted, if any, in order to make the branches identical, are colored. Unmodified nodes, on the other hand, are matched with their counterparts in the sibling tree and made white.

To make the above concrete, Figure 6 shows an example pair of plan trees arising in the plan diagram of Figure 2(a). In this figure, the white nodes represent the matching nodes, while the colored nodes represent the distinct nodes between the trees.

Plan Difference Metric. We now describe the procedure to quantify plan-tree differences. Our formulation uses $|T_i|$ and $|T_j|$ to represent the number of nodes in plan-trees T_i and T_j, respectively, and $|T_i \cap T_j|$ to denote the number of matching (white) nodes between the trees.

Now, ρ is measured as the classical Jaccard Distance [18] between the trees of the two plans, and is computed as

$$\rho(T_i, T_j) = 1 - \frac{|T_i \cap T_j|}{|T_i \cup T_j|} \tag{5}$$

While the above works for a pair of plans, we need to be able to extend the metric to handle an arbitrary set of plans, corresponding to the corners of the hyper-rectangle in the selectivity space. This is achieved through the following computation:

Given a set of n trees $\{T_1, T_2, \ldots, T_n\}$,

$$\rho(T_1, \ldots, T_n) = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \rho(T_i, T_j)}{\binom{n}{2}} \tag{6}$$

Note that the ρ values are normalized between 0 and 1, with values close to 0 indicating that all the plans are very similar to each other, and values close to 1 indicating that the plans are extremely dis-similar in structure.

Figure 8 depicts the ρ values calculated for a sample plan diagram after partitioning it into 20x20 squares. We see here that ρ reaches high values close to the origin and along the selectivity axes. This

GS_PQO (QueryTemplate Q, ErrorBound ϵ)

1. $\rho_t = \epsilon$
2. Optimize the points in the initial low-resolution grid.
3. Calculate the ρ plan density metric for each rectangle using Equation 6.
4. Organize the rectangles in a max-Heap structure based on their ρ values.
5. For the rectangle R_{top} at the top of the heap
6. If $\rho(R_{top}) \leq \rho_T$, stop
7. else
8. Extract R_{top} from the heap
9. Apply PQO interpolation to mid-points of qualifying edges of R_{top}.
   Optimize all the remaining mid-points.
10. Split R_{top} into four equal rectangles.
11. Compute ρ values for the smaller rectangles.
12. Insert the new rectangles into the heap
13. Return to 5
14. End Algorithm GS_PQO

Figure 7: The GS_PQO Algorithm

meshes perfectly with earlier observations in [11, 12, 14, 15, 16] that plans tend to be densely packed in precisely these regions of the selectivity space.

We now describe how GS_PQO utilizes the above characterization of plan-tree-differences. First, the grid sampling procedure is executed as mentioned earlier. Then, for each resulting rectangle, the ρ value is computed based on the plan-trees at the four corners, using Equation 6. The rectangles are organized in a max-Heap structure based on the ρ values, and the optimizations are directed towards the rectangle R_{top} at the top of the heap, i.e. with the current highest value of ρ. Specifically, the PQO principle is applied to the mid-points of all qualifying edges (those with common plans at both ends of the edge) in R_{top}, and all the remaining edge mid-points are explicitly optimized. The rectangle is then split into four smaller rectangles, for whom the ρ values are recomputed, and these rectangles are then inserted into the heap. This process continues until all the rectangles in the plan diagram have a ρ value that is below a threshold ρ_t. The threshold is a function of the ϵ bound given by the user, with lower thresholds corresponding to tighter bounds. Our empirical assessment suggests that setting the threshold to be equal to the error bound, i.e. $\rho_t = \epsilon$, is conducive to good performance.

The complete GS_PQO algorithm is shown in Figure 7.

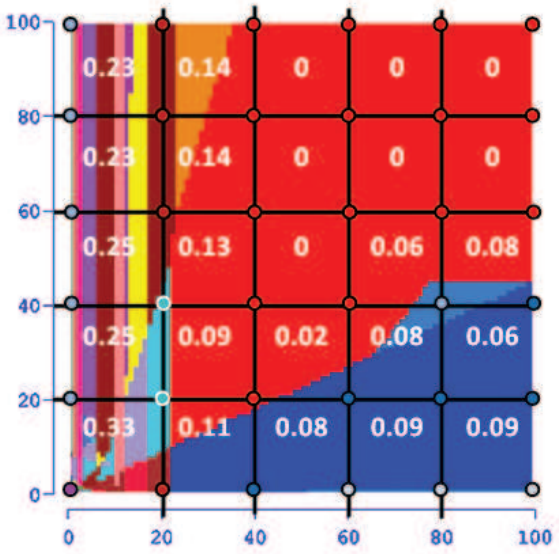


Figure 8: ρ calculated by GS_PQO

2.2 Class II Optimizers

In the algorithms described for the Class I optimizers, we run into situations wherein we are forced to pick from a set of equivalent candidate plans in order to make an assignment for a non-optimized query point. For example, in the RS_kNN approach, if there are multiple nearest neighbors at the same distance. Similarly, in the GS_PQO approach, when the ρ values of a rectangle goes below the threshold and there are unassigned points inside the region.

One obvious option in all the above cases is to make a random choice from the closest neighbouring plans. However, for Class II optimizers, which offer a "foreign plan costing" (FPC) feature, we can make a more informed selection: Specifically, cost all the candidate plans at the query point in question, and assign it the lowest cost plan. This method significantly helps in reducing the plan-location error, since it enables precise demarcation of the boundaries between plan optimality regions.

A point to be noted here is that plan-costing is much cheaper than the optimizer's standard optimal-plan-searching process [12], and hence the overheads incurred through costing are negligible compared to those incurred through optimization. In our experience, the overhead ratio of plan-costing to plan-searching is around **1:10** in the commercial optimizers, while in our implementation of this feature in PostgreSQL, it is close to **1:100**.

2.3 Class III Optimizers

The algorithms discussed thus far minimize the number of explicit optimizations performed by assuming certain properties of the plan diagram and using these properties to interpolate between the optimized query points. We now move on to presenting for the Class III optimizers, the DiffGen algorithm, which can be used to efficiently generate *completely accurate* plan diagrams. Subsequently, we provide

DiffGen (QueryTemplate QT)

1. Let A be an empty plan diagram.
2. Set $q = (0, 0)$
3. while($q \neq null$)
   (a) Optimize query Q at point q.
   (b) Let p_i and p_j be optimal and second-best plan at q, respectively.
   (c) For all points q' in the first quadrant of q
   if($c_i(q') \leq c_j(q)$), assign plan p_i to q'
   (d) Set q = next unassigned query point in A
4. **Return** A
5. End Algorithm DiffGen

Figure 9: The DiffGen Algorithm

a variant, the ApproxDiffGen algorithm, which trades error, based on the user's bound, for reduction in optimization effort. Both algorithms utilize the foreign-plan-costing (FPC) and plan-rank-list (PRL) features offered by the Class III optimizer API. Specifically, it is assumed that for each query point, the optimizer provides both the best plan and the second-best plan. As discussed in the Introduction, this is a feature that can be easily incorporated in today's systems with only marginal changes to the codebase.

2.3.1 The DiffGen Algorithm

The DiffGen algorithm for a 2D query template is shown in Figure 9. Let an optimization be performed at query point $q(x, y)$ in the selectivity space. Let p_i be the optimizer-estimated optimal plan at q, with cost $c_i(q)$, and let p_j be the *second best* plan, with cost $c_j(q)$. We then assign the plan p_i to all points q' in the *first quadrant* relative to q as the origin, which obey the constraint that $c_i(q') \leq c_j(q)$. After this step is complete, we then move to the next unassigned point in row-major order relative to q, and repeat the process, which continues until no unassigned points remain.

This algorithm is predicated on the *Plan Cost Motonicity* (PCM) assumption that the cost of a plan is monotonically non-decreasing throughout the selectivity space, which is true in practice for most query templates [9].

When the PCM property does not hold, we know that that that the cost function will still be monotonic along another quadrant [9]. The algorithm can be easily modified to take this into consideration. For example, if the costs are monotonically non-decreasing along the fourth quadrant, then the algorithm starts processing from the top-right of the plan diagram (Step 2),and the plan assignment is performed along the fourth quadrant (Step 3c).The quadrant in which the cost of a plan is non-decreasing can be easily obtained by comparing the costs of the plan at the 4 corners of the selectivity space.

The following theorem proves that the DiffGen algorithm will exactly produce the true plan diagram

P without any approximation whatsoever. That is, *by definition*, there is zero plan-identity and plan-location errors.

Theorem 1 *The plan assigned by DiffGen to any point in the approximate plan diagram* A *is exactly the same as that assigned in* P.

Proof: Let $P_o \subseteq \mathsf{P}$ be the set of points which were optimized. Consider a point $q' \in \mathsf{P} \setminus P_o$ with a plan p_i. Let $q \in P_o$ be the point that was optimized when q' was assigned the plan p_i. Let p_j be the second best plan at q.

For the sake of contradiction, let $p_k (i \neq k)$, be the optimal plan at q'. We know that for a cost-based optimizer, $c_k(q') < c_i(q')$. This implies that $c_k(q') < c_j(q)$ (due to the algorithm). Using the PCM property, we have $c_k(q) \leq c_k(q') \Rightarrow c_i(q) \leq c_k(q) < c_j(q)$. This means that p_j is not the second best plan at q, a contradiction. ■

2.3.2 The ApproxDiffGen Algorithm

While DiffGen always gives zero error, we now investigate the possibility of whether it is possible to utilize the permissible error bound of ϵ given by the user to reduce the computational overheads of DiffGen. To this end, we propose the following ApproxDiffGen algorithm: The plan assignment constraint $c_i(q') \leq c_j(q)$ is relaxed to be $c_i(q') \leq (1+\delta)c_j(q)$; $(\delta > 0)$, resulting in fewer optimizations being required to fully assign plans in the diagram. The choice of δ is a function of the user's ϵ error bound, and our empirical assessment indicates that setting $\delta = 0.1 * \epsilon$ is sufficient to both meet the error requirements and simultaneously significantly reduce the overheads. For example, $\epsilon = 10\%$ can be achieved with only around **1%** overheads.

3 Experimental Results

The testbed used in our experiments is the Picasso optimizer visualization tool [19], executing on a Sun Ultra 20 workstation equipped with an Opteron Dual Core 4GHz processor, 4 GB of main memory and 720 GB of hard disk, running the Windows XP Pro operating system. The experiments were conducted over plan diagrams produced from a variety of two, three, and four-dimensional **TPC-H** [24] and **TPC-DS** [25] based query templates. In our discussion, we use QT*x* to refer to a query template based on Query *x* of the TPC-H benchmark, and DSQT*x* to refer to a query template based on Query *x* of the TPC-DS benchmark. The TPC-H database was of size 1GB, while the TPC-DS database occupies 100GB. The plan diagrams were generated with a variety of industrial-strength database query optimizers – we present representative results here for a commercial optimizer anonymously referred to hereafter as OptCom, and a public-domain optimizer, hereafter referred to as OptPub.

In the remainder of this section, we evaluate the various approximaton plan diagram strategies with regard to their computational efficiency, given a user error bound for plan-identity and plan-locality. The two error-bounds we consider are 10% and 1% for both metrics.

Resolution / Dimension	Query Template	No. of Plans	Exhaustive Plan Diagram Generation Time	Approximation Time Taken		Optimizations Required (%)	
				RS_kNN	GS_PQO	RS_kNN	GS_PQO
100X100	QT2	44	0.5 hrs	10 mins (33%)	2 mins (9%)	33 %	7 %
	QT3	16	8 mins	3 mins (42%)	33 secs (7%)	42 %	2 %
	QT4	11	6 mins	1 min (12%)	21 secs (6%)	12 %	11 %
	QT5	23	0.75 hrs	6 mins (13%)	4 mins (9%)	13 %	7 %
	QT8	50	1 hr	31 mins (45%)	11 mins (16%)	45 %	11 %
	QT9	44	2 hrs	44 mins (40%)	15 mins (14%)	40 %	6 %
	QT10	17	12 mins	1 min (9%)	50 secs (7%)	9 %	4 %
	QT11	16	36 mins	3 mins (8%)	2 mins (7%)	8 %	5 %
	QT12	7	4 mins	22 secs (9%)	4 secs (2%)	9 %	4 %
	QT16	32	10 mins	2 mins (21%)	1 min (11%)	21 %	7 %
	QT20	33	4 hrs	1.5 hrs (40%)	16 mins (7%)	40 %	7 %
	QT21	42	0.5 hrs	3 mins (11%)	3 mins (11%)	11 %	4 %
300X300	QT2	76	9.6 hrs	2 hrs (23%)	20 mins (4%)	23 %	4 %
	QT3	22	1.7 hrs	9 mins (9%)	4 mins (4%)	9 %	4 %
	QT4	12	1 hr	5 mins (8%)	3 mins (5%)	8 %	5 %
	QT5	31	8.3 hrs	35 mins (7%)	15 mins (3%)	7 %	3 %
	QT8	92	10.5 hrs	19 mins (35%)	44 mins (3%)	35 %	3 %
	QT9	91	1 day 3 hrs	9 hrs (33%)	48 mins (3%)	33 %	3 %
	QT10	31	5 hrs	26 mins (10%)	4 mins (2%)	10 %	2 %
	QT11	20	2.5 hrs	20 mins (15%)	6 mins (4%)	15 %	4 %
	QT12	7	1 hr	1 min (2%)	2 mins (4%)	2 %	4 %
	QT16	38	1.6 hrs	5 mins (6%)	6 mins (6%)	6 %	6 %
	QT20	46	1 day 7 hrs	0.3 hrs (1%)	33 mins (4%)	1 %	4 %
	QT21	48	4.8 hrs	39.6 mins (14%)	8 mins (5%)	14 %	3 %
1000X1000	QT8	132	5 day 20 hrs	29 hrs (21%)	4.2 hrs (3%)	21 %	3%
	QT16	25	16 hrs	10 mins (1%)	10 mins (1%)	1%	1%
	QT21	58	2 day 6 hrs	2.7 hrs (5%)	32 mins (1%)	5 %	1%
100X100X100	QT8	190	6 day 10 hrs	24 hrs (16%)	2.4 hrs (7%)	16 %	7%
	QT9	404	10 day	64 hrs (27%)	24 hrs (10%)	27 %	10%
	QT21	130	3 day	7.6 hrs (11%)	3.5 hrs (5%)	11 %	5%
30X30X30X30	QT8	243	5 days	23 hrs (19%)	12 hrs (10%)	19 %	10%

Table 3: Algorithm Efficiency for Class I optimizers with TPC-H database ($\epsilon = 10\%$)

Resolution / Dimension	Query Template	No. of Plans	Exhaustive Plan Diagram Generation Time	Approximation Time Taken		Optimizations Required (%)	
				RS_kNN	GS_PQO	RS_kNN	GS_PQO
100X100	DSQT 12	13	16 mins	4 mins (25%)	1 min (5%)	25 %	5 %
	DSQT 17a	39	6.7 hrs	2.6 hrs (39%)	40 mins (10%)	39 %	10 %
	DSQT 18	47	2.6 hrs	1.5 hrs (53%)	14mins (9%)	53 %	9 %
	DSQT 19	36	2 hrs	19 mins (18%)	7 mins (7%)	18 %	7 %
	DSQT 25	33	7 hrs	4.6 hrs (65%)	46 mins (11%)	65 %	11 %
	DSQT 25a	51	6.5 hrs	1.5 hrs (24%)	42 mins (11%)	24 %	11 %
	DSQT 25b	45	7.3 hrs	2.6 hrs (36%)	48 mins (11%)	36 %	11 %
300X300	DSQT 12	15	2.2 hrs	37 mins (29 %)	5 mins (4 %)	29 %	4 %
	DSQT 18	81	22.5 hrs	8.7 hrs (38%)	2.3 hrs (10%)	38 %	10 %
	DSQT 19	42	16.2 hrs	1 hr (7%)	58 mins (6%)	7 %	6 %

Table 4: Algorithm Efficiency for Class I optimizers with TPC-DS database ($\epsilon = 10\%$)

3.1 Class I Optimizers

We start with evaluating the performance of the two algorithms applicable to Class I optimizers, namely, RS_kNN and GS_PQO. In the RS_kNN algorithm, as mentioned earlier, the parameter δ, which specifies the transition of the algorithm from Stage 1 to Stage 2, is set to 0.3. For the GS_PQO algorithm, the resolution of the initial grid is set to $r_0 = (0.1 \times r)^D$, where r is the resolution at which the plan diagram is to be generated, and D is the dimensionality of the selectivity space.

For the above framework, Table 3 shows the algorithmic efficiency of the RS_kNN and GS_PQO algorithms relative to the brute-force exhaustive approach for a variety of 2D, 3D and 4D query tem-

Resolution/ Resolution	Query Template	No. of Plans	Exhaustive Plan Diagram Generation time	Time taken by GS_PQO	Optimizations performed by GS_PQO (%)
100X100	QT 8	50	1.1 hrs	28min (40%)	40 %
	QT 9	44	2 hrs	26min (24%)	24 %
300X300	QT 8	92	10.5 hrs	3.6 hrs (35%)	35 %
	QT 9	91	1 day	7.3 hrs (30%)	30 %
1000X1000	QT 8	132	5 days 20 hrs	1 day 18 hrs (30%)	30 %
100X100X100	QT 8	190	6 days 10 hrs	1 day 14 hrs (25%)	25 %
	QT 9	404	10 days	4 days (40%)	40 %
	QT 21	130	2 days 22 hrs	11 hrs (16%)	16 %

Table 5: Algorithm Efficiency for Class I optimizers with TPC-H database ($\epsilon = 1\%$)

Resolution/ Resolution	Query Template	No. of Plans	Exhaustive Plan Diagram Generation time	Time taken by GS_PQO	Optimizations performed by GS_PQO (%)
100X100	DSQT 17a	39	6.7 hrs	2 hrs (35%)	35 %
	DSQT 18	47	2.6 hrs	1 hr (40 %)	40 %
	DSQT 19	36	2 hrs	1.1 hrs (35 %)	35 %
	DSQT 25	33	7 hrs	2 hrs (27%)	27 %
	DSQT 25a	51	6.5 hrs	2.1 hrs (30%)	30 %
	DSQT 25b	45	7.3 hrs	2.5 mins (33%)	33 %
300X300	DSQT 18	81	22.5 hrs	9 hrs (40%)	40 %
	DSQT 19	42	16.2 hrs	4.7 hrs (29%)	29 %

Table 6: Algorithm Efficiency for Class I optimizers with TPC-DS database ($\epsilon = 1\%$)

plates, with a user error bound of 10%. The efficiency is presented both in terms of actual time, as well as in terms of the number of optimizations that were carried out. The bracketed numbers in the *TimeTaken* columns indicate the percentage time taken relative to the exhaustive approach.

We see in Table 3 that the RS_kNN algorithm requires a substantial amount of time, or equivalently, number of optimizations, to generate the approximate plan diagram. For example, with the 3D QT9 template at a 1000 resolution, RS_kNN takes about 27% of the exhaustive time. On the other hand, when we consider GS_PQO, we see that it has a much better performance, requiring not more than 15% even in the worst-case across all templates.

Turning our attention to Table 4, which repeats the above experiment on the TPC-DS database, we see that the results are even more striking. RS_kNN consumes very large overheads in general, whereas GS_PQO again does not exceed 15%.

An interesting point to note in both these tables is that the optimization percentages are virtually identical to the time percentages. This means that the interpolation mechanisms of kNN and PQO take insignificant time as compared to making optimizer calls.

When the user's error constraint is tightened from 10 percent to 1 percent, the resulting algorithmic performance is shown in Table 5 and Table 6. Only GS_PQO is shown since for this stringent constraint, the RS_kNN algorithm tends to optimize close to the entire space. It can be seen from the table that by optimizing only around 40% of the points, GS_PQO is able to generate extremely high-quality approximate plan diagrams.

To demonstrate that the above results are not specific to OptCom, a sample comparison of the algorithms across other commercial optimizers, anonymously referred to as OptA, OptB and OptC, is given in Table 7. We see here that for different query templates (2D, resolution 100) with an error bound of 10%, GS_PQO again incurs only low overheads as compared to RS_kNN.

Resolution / Dimension	Query Template	No. of Plans	Exhaustive Plan Diagram Generation Time	Approximation Time Taken		Optimizations Required (%)	
				RS_kNN	GS_PQO	RS_kNN	GS_PQO
OptA	2	10	4 hrs	14 mins (6%)	10 mins (4%)	6 %	4 %
	5	16	3 hrs 18 mins	1 hr (30%)	14 mins (7%)	30 %	7 %
	7	13	3 hrs 43 mins	22 mins (10%)	14 mins (6.5%)	10 %	6.5 %
	8	74	7 hrs 41 mins	2.6 hrs (35%)	1.2 hr (15%)	35 %	15 %
	9	47	5 hrs 20 mins	2.2 hrs (42%)	41 mins (13%)	42 %	13 %
	21	21	3 hrs 40 mins	19 mins (9%)	4 mins (2%)	9 %	2 %
OptB	2	20	38 mins	5 mins (12%)	3 mins (7%)	12 %	7 %
	5	12	3 mins	4 secs (2%)	2 secs (1%)	2 %	1 %
	7	6	11 mins	26 secs (4%)	26 secs (4%)	4 %	4 %
	8	12	10 mins	1 min (13%)	36 secs (6%)	13 %	6 %
	9	18	14 mins	2 min (16%)	50 secs (6%)	16 %	6 %
	21	8	3 mins	2 min (60%)	20 secs (11%)	60 %	11 %
OptC	2	12	22 mins	1 min (5%)	13 secs (1%)	5 %	1 %
	5	3	25 mins	30 secs (2%)	15 secs (1%)	2 %	1 %
	7	10	16 mins	28 secs (3%)	10 secs (1%)	3 %	1 %
	8	16	21 mins	2 mins (10%)	51 secs (4%)	10 %	4 %
	9	4	20 mins	36 secs (3%)	12 secs (1%)	3 %	1 %
	21	10	35 mins	4 mins (12%)	1.75 mins (5%)	12 %	5 %

Table 7: Comparative Study of Approximation Techniques for different DB-Engine

3.2 Class II Optimizers

We now move on to demonstrate how the FPC feature, provided by Class II optimizers, can be used to improve the performance of GS_PQO. Tables 8 and 9 show the effort required by GS_PQO for obtaining approximate plan diagrams with an error bound of 10% on the TPC-H and TPC-DS benchmarks, respectively. We see here that GS_PQO consistently completes the approximation in less than 10% time, or equivalently, optimizations, testifying to the utility of FPC in improving the performance.

With a error bound of 1%, however, the role of FPC becomes limited since interpolation is at a premium, and therefore the diagram generation time is similar to that seen for Class I optimizers.

3.3 Class III Optimizers

Turning our attention to Class III optimizers, we now evaluate the two algorithms, DiffGen and ApproxDiffGen. For this experiment, the OptPub engine was modified to (a) implement the FPC feature internally, and (b) to return the second best plan along with the optimal plan when the "explain" command is executed.

As can be seen in Table 10, DiffGen usually requires at most 10% optimizations to generate a *completely accurate* plan diagram for all query templates, except those based on Query 8, the reason for which is discussed below. The good performance of DiffGen can be attributed to the following: Along with the optimizations being performed at select points, all points (except the origin) are costed exactly once. Since the FPC feature is internalized in the optimizer, the overhead incurred is very small, and an important byproduct of this minor investment is the ability to also obtain the cost diagram corresponding to the plan diagram.

Though 10% optimizations is usually the order of the day, there are occasional scenarios when the DiffGen algorithm requires a substantial number of optimizations to generate the plan diagram. Such a situation is seen for QT8 – the reason is that the cost of the second best plan is very close to that of the optimal plan over an extended region. Even though the plan switch occurs much later, this close-to-optimal cost causes the algorithm to optimize at frequent intervals as the constraint $c_i(q') \leq c_j(q)$ is

Resolution/ Dimension	Query Template	No. of Plans	Exhaustive Plan Diagram Generation time	Time taken by GS_PQO	Optimizations performed by GS_PQO (%)
100X100	QT2	44	0.5 hrs	2 mins (7%)	7 %
	QT3	16	8 mins	28 secs (6%)	6 %
	QT4	11	6 mins	18 secs (5%)	5 %
	QT5	23	0.75 hrs	3 mins (7%)	7 %
	QT8	50	1 hr	5 mins (7.5%)	7.5 %
	QT9	44	2 hrs	8 mins (7.5%)	7.5 %
	QT10	17	12 mins	43 secs (6%)	6 %
	QT11	16	36 mins	1 min (3%)	3 %
	QT12	7	4 mins	4 secs (2%)	2 %
	QT16	32	10 mins	1 min (10%)	10 %
	QT20	33	4 hrs	8 mins (3.5%)	3.5 %
	QT21	42	0.5 hrs	2 mins (8%)	8 %
300X300	QT2	76	9.6 hrs	11 mins (2%)	2%
	QT3	22	1.7 hrs	3 mins (3%)	3 %
	QT4	12	1 hr	2 mins (3%)	3 %
	QT5	31	8.3 hrs	10 mins (2%)	2 %
	QT8	92	10.5 hrs	12 mins (2%)	2 %
	QT9	91	26.8 hrs	32 mins (1.8%)	1.8 %
	QT10	31	4.5 hrs	5 mins (1.6%)	1.6 %
	QT11	20	2.5 hrs	5 mins (3.5%)	3.5 %
	QT12	7	1 hr	30 secs (1%)	1 %
	QT16	38	1.6 hrs	3 mins (3%)	3 %
	QT20	46	1 day 7.5 hrs	28 mins (1.8%)	1.8 %
	QT21	48	4.8 hrs	4 mins (1%)	1 %
1000X1000	QT8	132	6 days	3.8 hrs (3%)	3 %
	QT16	25	16 hrs	9 mins (1%)	1 %
	QT21	58	2 days 6 hrs	32 mins (1%)	1 %
100X100X100	QT8	190	6 day 10 hrs	hrs (4%)	4 %
	QT9	404	10 days	21.6 hrs (9%)	9 %
	QT21	130	3 days	3.5 hrs (5%)	5 %
30X30X30X30	QT8	243	5 days	12 hrs (10%)	10 %

Table 8: Efficiency of GS_PQO for Class II optimizers with TPC-H database ($\epsilon = 10\%$)

Resolution/ Dimension	Query Template	No. of Plans	Exhaustive Plan Diagram Generation time	Time taken by GS_PQO	Optimizations performed by GS_PQO (%)
100X100	DSQT 12	13	6.7 hrs	8 mins (2%)	2 %
	DSQT 17a	39	6.7 hrs	20 mins (5%)	5 %
	DSQT 18	47	2 hr 36 min	4 mins (5%)	5 %
	DSQT 19	36	1 hr 48 min	3 mins (4%)	4 %
	DSQT 25	33	7 hrs	45 mins (10%)	10 %
	DSQT 25a	51	6.5 hrs	42 mins (10%)	10 %
	DSQT 25b	45	7.3 hrs	30 mins (7%)	7 %
300X300	DSQT 12	15	2 hr 11 min	30 mins (4%)	4 %
	DSQT 18	81	22.5 hrs	1.2 hrs (5%)	5 %
	DSQT 19	42	16.2 hrs	24 mins (3%)	3 %

Table 9: Efficiency of GS_PQO for Class II optimizers with TPC-DS database ($\epsilon = 10\%$)

Resolution Dimension	Query Template	No. of Plans	Exhaustive Plan Diagram Generation time	Time taken by DiffGen	Optimizations performed by DiffGen (%)
1000×1000	QT5	22	5 hrs 20 mins	4 mins (1%)	0.17 %
	QT8	20	6 hrs 10 mins	2 hrs 47 mins(45%)	44 %
	QT9	16	6 hrs 40 mins	40 mins (10%)	7.4 %
$100 \times 100 \times 100$	QT5	23	5 hrs 48 mins	13mins (3%)	2.4 %
	QT8	49	5 hrs 58 mins	2 hrs 2 mins (34%)	32 %
	QT9	22	6 hrs 45 mins	5 mins (1%)	0.24 %
$30 \times 30 \times 30 \times 30$	QT5	37	4 hrs 50 mins	25 mins(8%)	5.8 %
	QT8	28	4 hrs 30 mins	1 hr 18 mins(29%)	26 %
	QT9	62	6 hrs 10 mins	7 mins(2%)	0.7 %

Table 10: Performance of the DiffGen Algorithm

Resolution Resolution	Query Template	No. of Plans	Exhaustive Plan Diagram Generation time	Approximate Plan Diagram Generation time	Optimizations performed by ApproxDiffGen (%)
1000×1000	QT5	22	5 hrs 20 mins	3 mins (1%)	0.09 %
	QT8	20	6 hrs 10 mins	9 mins (2%)	1.16 %
	QT9	16	6 hrs 40 mins	4 mins (1%)	0.12 %
$100 \times 100 \times 100$	QT5	23	5 hrs 48 mins	10 mins (3%)	1.7 %
	QT8	49	5 hrs 58 mins	17 mins (5%)	3.4 %
	QT9	22	6 hrs 45 mins	4 mins (1%)	0.06 %
$30 \times 30 \times 30 \times 30$	QT5	37	4 hrs 50 mins	20 mins (7%)	4.5 %
	QT8	28	4 hrs 30 mins	10 mins (4%)	1.9 %
	QT9	62	6 hrs 10 mins	5 mins (1%)	0.3 %

Table 11: Performance of the ApproxDiffGen Algorithm ($\epsilon = 10\%$)

easily violated.

Turning our attention to the ApproxDiffGen algorithm, we find that it can be consistently used to generate an approximate plan diagram with a 10% error bound, while performing less than 5% optimizations – as highlighted in Table 11. A point to note here is that, even for QT8, due to the relaxation of the effect of the proximity of the second best plan, the plan diagram is now obtained incurring only a low overhead.

A related point to note is that unlike the Optimizer I and II classes where the time and optimization overheads were virtually identical, here the time overheads are a little more than that of optimization. The reason is that, although FPC is very cheap, since it has to be invoked for a very large number of points, it adds a small but perceptible time overhead.

4 Conclusions

We have investigated in this paper the efficient generation of approximate plan diagrams, a key resource in the analysis and redesign of modern database query optimizers. Based on the optimizer's API capabilities, we made a partitioning into three different classes of optimizers, and developed appropriate approximation techniques for each class. For Class I, which only provides the optimal plan, our experimental results showed that the GS_PQO algorithm, which combines grid sampling with PQO interpolation at the micro level, performed very adequately requiring less than 15% overheads as compared to the exhaustive approach, for an error bound of 10%. These overheads came down to 10% when the same algorithm was used in Class II optimizers, due to their additional FPC feature. Finally, for Class III systems, we proved that the DiffGen algorithm produced zero errors and was able to do so incurring overheads of less than 10%. However, it performs poorly for query templates that have the second-best plan being very close to the optimal choice over an extended region. Finally, the ApproxDiffGen algorithm traded error for performance, and was able to satisfy the 10% error bound with less than 5% optimizations. It was also able to adequately handle the problem templates of DiffGen.

In summary, our work has shown that it is indeed possible to efficiently generate high-quality approximations to high-dimension and high-resolution plan diagrams, with typical overheads being an *order of magnitude lower* than the brute-force approach. We hope that our results will encourage all database vendors to incorporate the foreign-plan-costing and plan-rank-list features, both of which were critical to the excellent performance of DiffGen and ApproxDiffGen, in their optimizer APIs.

References

[1] G. Antonshenkov, "Dynamic Query Optimization in Rdb/VMS", *Proc. of 9th IEEE Intl. Conf. on Data Engineering (ICDE)*, April 1993.

[2] M. Charikar, S. Chaudhuri, R. Motwani and V. Narasayya,"Towards Estimation Error Guarantees for Distinct Values", *Proc. of ACM Symp. on Principles of Database Systems (PODS)*, 2000.

[3] 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.

[4] 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.

[5] A. Ghosh, J. Parikh, V. Sengar and J. Haritsa, "Plan Selection based on Query Clustering", *Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2002.

[6] R. Gonzalez and R. Woods, *Digital Image Processing*, Pearson Prentice Hall, 2007.

[7] P. Haas and L. Stokes. "Estimating the number of classes in a finite population". *In Journal of the American Statistical Association, 93*,1998.

[8] P. Haas, J. Naughton, S. Seshadri and L. Stokes,"Sampling-Based Estimation of the Number of Distinct Values of an Attribute", *Proc. of 21st Intl. Conf. on Very Large Databases (VLDB)*, 1995.

[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", *Tech. Rep. TR-2007-02, DSL/SERC, Indian Inst. of Science*, 2007. `http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2007-02.pdf`

[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] V. Prasad, "Parametric Query Optimization: A Geometric Approach", *Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur*, April 1999.

[15] S. Rao, "Parametric Query Optimization: A Non-Geometric Approach", *Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur*, March 1999.

[16] 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.

[17] N. Reddy, "Next-Generation Relational Query Optimizers", *Master's Thesis, Dept. of CSA, Indian Institute of Science*, June 2005, `http://dsl.serc.iisc.ernet.in/publications/thesis/naveen.pdf`.

[18] P. Tan, M. Steinbach and V. Kumar, *Introduction to Data Mining*, Addison-Wesley, 2005.

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

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

[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://postgresql.org`

[24] `http://www.tpc.org/tpch`

[25] `http://www.tpc.org/tpcds`

5 Appendix

In this section we have listed the n-dimensional version of the algorithms described and also the implementation procedure of different techniques mentioned earlier in the paper.

5.1 RS_kNN

We describe the implementation procedure of the NN-Interpolation and Low Pass Filtering technique mentioned so far.

5.1.1 NN-Interpolation

As we have mentioned earlier the interpolation technique relies on the nearest-neighbor approach to look for a suitable plan around a non-sampled point as a candidate plan for it. The algorithm listed in the Figure 10 is invoked for each non sampled point.

From here onwards we have used d to represent the dimension of the diagram. The variable $dist$ is used to set the chessboard distance at which we are interested in finding the neighbors e.g. if $dist = 4$ then the function *recursiveNN()* derives all possible offsets required to find out neighbors at that particular chessboard distance. The variable $dimPresent$ is used to avoid generating offsets for neighbors at a lesser distance i.e. if $dist = 4$ then we should not generate neighbors for $dist = 3, 2 or 1$. This is ensured by making at least one of the coordinates of the offset equal to $dist$. $dimPresent$ is used to implement the same by forcefully turning the lowest dimension to $dist$ if none of the higher dimension is set to so. Then we add these offsets with the coordinates of non-sampled point X and apply the interpolation technique thereafter.

5.1.2 Low-Pass

We run one iteration of Low-Pass Filter on the approximate diagram to remove jagged edges introduced by NN interpolation. We look at all the neighbors at distance 1 from a non-sampled point. This can be done by invoking the *kNearestNeighbor* algorithm illustrated in Figure 10 with $dist = 1$. If any of the neighboring plans occupies more than 50% of points, we assign that plan to the non-sampled point.

5.2 GS_PQO

The n-dimensional GS_PQO algorithm is almost same as described in Section 2.1.2 except the *initial grid sampling* and *rectangle decomposition*. The initial grid sampling employs a simple recursive function *InitialGSPQO* shown in Figure 11 to optimize the corner points of initial rectangles. In the rectangle decomposition step we need to optimize or interpolate the mid-points of all the $d \cdot 2^{d-1}$ edges of a d dimensional hyper-rectangle and break it into 2^d equal hyper-rectangles. The complete algorithm is illustrated in Figure 11.

```
//Global variables
Non-Sampled point: X(x_1, x_2, ..., x_d);
Dimension: d;
Distance: dist;
Array of length d: DimVar;
boolean dimPresent;
Queue Q;
kNearestNeighbor()

1. dist = 1;
2. Call recursiveNN(d); //Recursively check neighbors at distance dist
3. if Q is not NULL
4.     Assign plan P occupying maximum points to X.
5.     Return;
6. dist + +;
7. Go to Step 2;

recursiveNN(Depth)

1. if Depth = 1
2.     if dimPresent = true,
3.         for i= -dist to +dist, increment i by 1
4.             dimVar[1] = i;
5.             doNNJob();
6.     else
7.         dimVar[1] = -dist;
8.         doNNJob();
9.         dimVar[1] = +dist;
10.        doNNJob();
11. else
12.    for i= -dist to +dist, increment i by 1
13.        dimVar[Depth] = i;
14.        dimPresent = false;
15.        if i = -dist or i = +dist
16.            dimPresent = true;
17.        recursiveNN(Depth-1);

doNNJob()

1. NN[1...d] : dimvar[1...d] + X[1...d]
2. if NN[1...d] is a sampled point
3.     Add NN into Queue Q;
4. Return;
```

Figure 10: The n-Dimensional RS_kNN Interpolation Algorithm

//Global Variables
Integer $interval$; //Initial GSPQO interval
Array of length d: $DimVar$;
Resolution: res;
GS_PQO (QueryTemplate Q, ErrorBound ϵ,Dimension d)

1. $\rho_t = \epsilon$
2. $InitialGSPQO(d)$; //Optimize points in the initial low-resolution grid
3. Calculate the ρ plan density metric for each hyper-rectangle with 2^d corners using Equation 6.
4. Organize the hyper-rectangles in a max-Heap structure based on their ρ values.
5. For the hyper-rectangle R_{top} at the top of the heap
6. If $\rho(R_{top}) \leq \rho_T$ stop
7. else
8. Extract R_{top} from the heap
9. Apply PQO interpolation to the mid-points of qualifying edges of R_{top}. Optimize all the remaining mid-points.
10. Split R_{top} into 2^d equal hyper-rectangles.
11. Compute ρ values for the smaller hyper-rectangles.
12. Insert the new hyper-rectangles into the heap
13. Return to 5
14. End Algorithm GS_PQO

InitialGSPQO(Depth d)

1. if Depth = 1
2. For DimVar[1] = 1 to res increment by 1
3. Optimize the point DimVar[1 ... d];
4. else
5. For DimVar[d] = 1 to res increment by 1
6. InitialGSPQO($d-1$);

Figure 11: The n-Dimensional GS_PQO Algorithm