Query Processing without Estimation

Anshuman Dutt Jayant R. Haritsa

Technical Report
TR-2014-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

(This report replaces the earlier version TR-2013-01)

Abstract

Selectivity estimates for optimizing OLAP queries often differ significantly from those actually encountered during query execution, leading to poor plan choices and inflated response times. We propose here a conceptually new approach to address this problem, wherein the compile-time estimation process is completely eschewed for error-prone selectivities. Instead, a small "bouquet" of plans is identified from the set of optimal plans in the query's selectivity error space, such that at least one among this subset is near-optimal at each location in the space. Then, at run time, the actual selectivities of the query are incrementally "discovered" through a sequence of partial executions of bouquet plans, eventually identifying the appropriate bouquet plan to execute. The duration and switching of the partial executions is controlled by a graded progression of isocost surfaces projected onto the optimal performance profile. We prove that this construction results in bounded overheads for the selectivity discovery process and consequently, guaranteed worst-case performance. In addition, it provides repeatable execution strategies across different invocations of a query.

The plan bouquet approach has been empirically evaluated on both PostgreSQL and a commercial DBMS, over the TPC-H and TPC-DS benchmark environments. Our experimental results indicate that, even with conservative assumptions, it delivers substantial improvements in the worst-case behavior, without impairing the average-case performance, as compared to the native optimizers of these systems. Moreover, the bouquet technique can be largely implemented using existing optimizer infrastructure, making it relatively easy to incorporate in current database engines.

Overall, the bouquet approach provides novel guarantees that open up new possibilities for robust query processing.

1 Introduction

Cost-based database query optimizers estimate a host of *selectivities* while identifying the ideal execution plan for declarative OLAP queries. For example, consider **EQ**, the simple SPJ query shown in Figure 1 for enumerating orders of cheap parts – here, the optimizer estimates the selectivities of a selection predicate (*p_retailprice*) and two join predicates (*part* $\bowtie$ *lineitem*, *lineitem* $\bowtie$ *orders*). In practice, these estimates are often significantly in error with respect to the actual values subsequently encountered during query execution. Such errors, which can even be in *orders of magnitude* in real database environments [17], arise due to a variety of well-documented reasons [22], including outdated statistics, attribute-value independence(AVI) assumptions, coarse summaries, complex user-defined predicates, and error propagation in the query execution operator tree [15]. Moreover, in environments such as ETL workflows, the statistics may actually be *unavailable* due to data source constraints, forcing the optimizer to resort to "magic numbers" for the values (e.g. $1/10$ for equality selections [21]). The net result of these erroneous estimates is that the execution plans recommended by the query optimizer may turn out to be poor choices at run-time, resulting in substantially inflated query response times.

```
select * from lineitem, orders, part
where p_partkey = l_partkey and l_orderkey =
      o_orderkey and p_retailprice < 1000
```

Figure 1: **Example Query (EQ)**

A considerable body of literature exists on proposals to tackle this classical problem. For instance, techniques for improving the *statistical quality* of the meta-data include improved summary struc-

tures [1, 18], feedback-based adjustments [22], and on-the-fly re-optimization of queries [16, 4, 19]. A complementary approach is to identify *robust plans* that are relatively less sensitive to estimation errors [9, 3, 4, 13]. While these prior techniques provide novel and innovative formulations, they are limited in their scope and performance, as explained later in the related work section.

Plan Bouquet Approach

In this paper, we investigate a conceptually new approach, wherein the compile-time estimation process is completely eschewed for error-prone selectivities. Instead, these selectivities are systematically *discovered* at run-time through a calibrated sequence of cost-limited plan executions. In a nutshell, we attempt to side-step the selectivity estimation problem, rather than address it head-on, by adopting a *"seeing is believing"* viewpoint on these values.

1D Example We introduce the new approach through a restricted 1D version of the EQ example query wherein only the p_retailprice selection predicate is error-prone. First, through repeated invocations of the optimizer, we identify the "parametric optimal set of plans" (POSP) that cover the entire selectivity range of the predicate. A sample outcome of this process is shown in Figure 2, wherein the POSP set is comprised of plans P1 through P5. Further, each plan is annotated with the selectivity range over which it is optimal – for instance, plan P3 is optimal in the (1.0%, 7.5%] interval. (In Figure 2, P = Part, L = Lineitem, O = Order, NL = Nested Loops Join, MJ = Sort Merge Join, and HJ = Hash Join) .

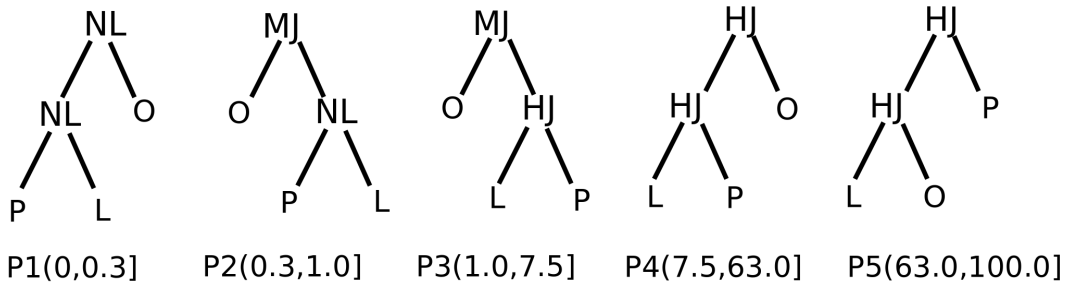

Figure 2: POSP plans on p_retailprice dimension

The optimizer-generated costs of these POSP plans over the selectivity range are shown (on a log-log scale) in Figure 3. On this figure, we first construct the "POSP infimum curve" (**PIC**), defined as the trajectory of the minimum cost from among the POSP plans – this curve represents the ideal performance. The next step, which is a distinctive feature of our approach, is to *discretize* the PIC by projecting a graded progression of *isocost* (IC) steps onto the curve. For example, in Figure 3, the dotted horizontal lines represent a geometric progression of isocost steps, IC1 through IC7, with each step being *double* the preceding value. The intersection of each IC with the PIC (indicated by ■) provides an associated selectivity, along with the identity of the best POSP plan for this selectivity. For example, in Figure 3, the intersection of IC5 with the PIC corresponds to a selectivity of 0.65% with associated POSP plan P2. We term the subset of POSP plans that are associated with the intersections as the "plan bouquet" for the given query – in Figure 3, the bouquet consists of {P1, P2, P3, P5}.

The above exercises are carried out at query compilation time. Subsequently, at run-time, the correct query selectivities are explicitly discovered through a sequence of *cost-limited* executions of bouquet plans. Specifically, beginning with the cheapest isocost step, we iteratively execute the bouquet plan assigned to each step until either:

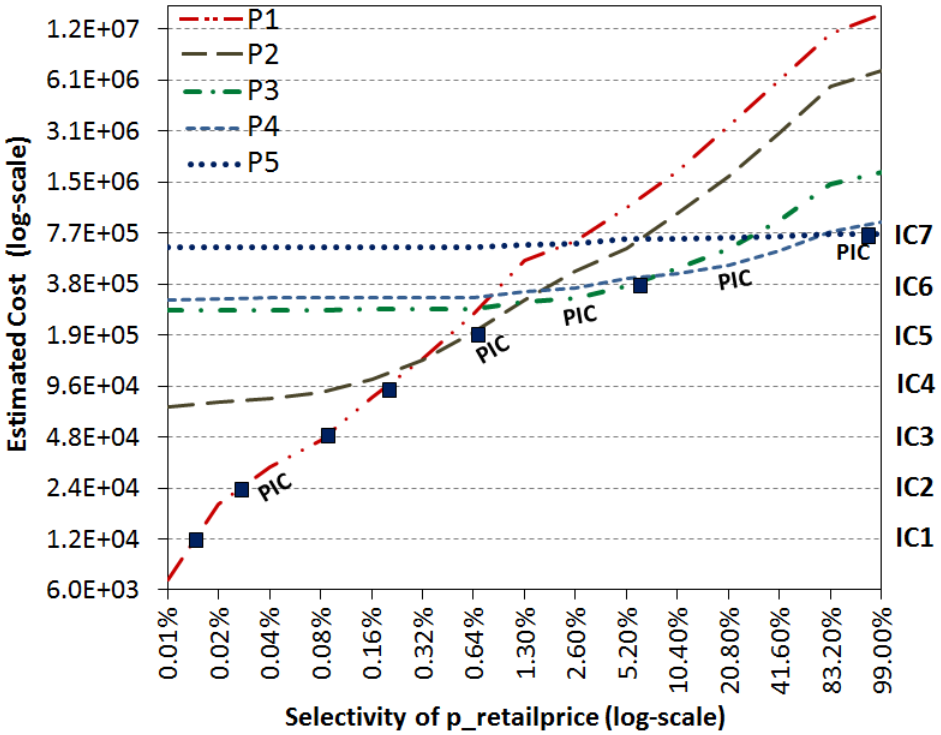

Figure 3: POSP performance (log-log scale)

1. The partial execution overheads exceed the step's cost value – in this case, we know that the actual selectivity location lies beyond the current step, motivating a switch to the next step in the sequence; or
2. The current plan completes execution within the budget – in this case, we know that the actual selectivity location has been reached, and a plan that is at least 2-competitive wrt the ideal choice was used for the final execution.

Example To make the above process concrete, consider the case where the selectivity of p_retailprice is 5%. Here, we begin by partially executing plan P1 until the execution overheads reach IC1 (1.2E4 | 0.015%). Then, we extend our cost horizon to IC2, and continue executing P1 until the overheads reach IC2 (2.4E4| 0.03%), and so on until the overheads reach IC4 (9.6E4 | 0.2%). At this juncture, there is a change of plan to P2 as we look ahead to IC5 (1.9E5 | 0.65%), and during this switching all the intermediate results (if any) produced thus far by plan P1 are *jettisoned*. The new plan P2 is executed till the associated overhead limit (1.9E5) is reached. The cost horizon is now extended to IC6 (3.8E5 | 6.5%), in the process jettisoning plan P2's intermediate results and executing plan P3 instead. In this case, the execution will complete before the cost limit is reached since the actual location, 5%, is less than the selectivity limit of IC6. Viewed in toto, the net sub-optimality turns out to be 1.78 since the exploratory overheads are 0.78 times the optimal cost, and the optimal plan itself was (coincidentally) used for the final execution.

Extension to Multiple Dimensions When the above approach is generalized to the multi-dimensional selectivity environment, the IC steps and the PIC curve become surfaces, and their intersections represent selectivity surfaces on which multiple bouquet plans may be present. For example, in the 2-D case, the IC steps are horizontal planes cutting through a hollow 3D PIC surface, typically

resulting in hyperbolic intersection contours with different plans associated with disjoint segments of this contour – an instance of this scenario is shown in Figure 7.

Notwithstanding these changes, the basic mechanics of the bouquet algorithm remain virtually identical. The primary difference is that we jump from one IC surface to the next only after it is determined (either explicitly or implicitly) that *none* of the bouquet plans present on the current IC surface can completely execute the given query within the associated cost budget.

Performance Characteristics

At first glance, the plan bouquet approach, as described above, may appear to be utterly absurd and self-defeating because: (a) At compile time, considerable preprocessing may be required to identify the POSP plan set and the associated PIC; and (b) At run-time, the overheads may be hugely expensive since there are multiple plan executions for a single query – in the worst scenario, as many plans as are present in the bouquet!

However, we will attempt to make the case in the remainder of this paper, that it is indeed possible, through careful design, to have *plan bouquets efficiently provide robustness profiles that are markedly superior to the native optimizer's profile*. Specifically, if we define robustness to be the worst-case sub-optimality in plan performance that can occur due to selectivity errors, the bouquet mechanism delivers substantial robustness improvements, while providing comparable or improved average-case performance.

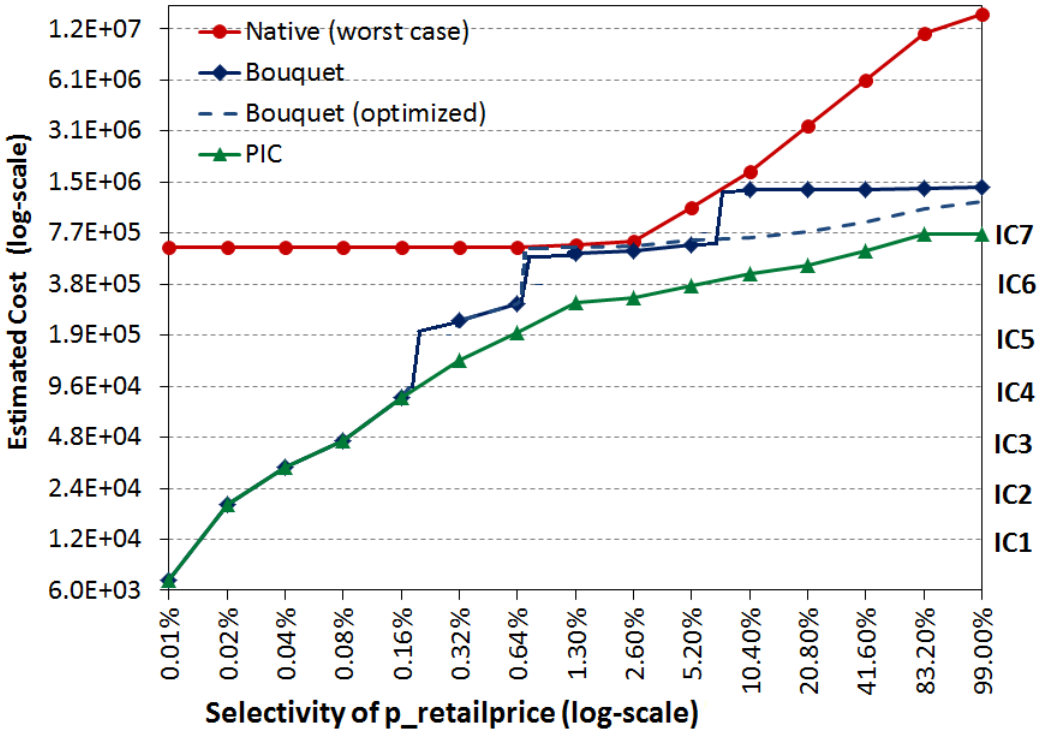

Figure 4: Bouquet Performance (log-log scale)

For instance, the runtime performance of the bouquet technique on EQ is profiled in Figure 4 (dark blue curve). We observe that its performance is much closer to the PIC (dark green) as compared to the worst case profile for the native optimizer (dark red), comprised of the supremum of the individual plan profiles. In fact, the worst case sub-optimality for the bouquet is only 3.6 (at 6.5%), whereas the native

optimizer suffers a sub-optimality of around 100 when P5 (which is optimal for large selectivities) is mistakenly chosen to execute a query with a low selectivity of 0.01%. The average sub-optimality of the bouquet, computed over all possible errors, is 2.4, somewhat worse than the 1.8 obtained with the native optimizer. However, when the enhancements described later in this paper are incorporated, the optimized bouquet's performance (dashed blue) improves to 3.1 (worst case) and 1.7 (average case), thereby dominating the native optimizer on both metrics.

Our motivation for the cost-based discretization of the PIC is that it leads to *guaranteed* bounds on worst-case performance. For instance, we prove that the cost-doubling strategy used in the 1D example results in an *upper-bound of 4* for the worst-case sub-optimality – this bound is inclusive of all exploratory overheads incurred by the partial executions, and is irrespective of the query's actual selectivity. In fact, we can go further to show that 4 is the best competitive factor achievable by *any* deterministic algorithm. For the multi-dimensional case, the bound becomes 4 times the bouquet cardinality (more accurately, the plan cardinality of the densest contour), and we present techniques to limit this cardinality to a small value. To our knowledge, these robustness bounds are the first such guarantees to be presented in the database literature (although similar characterizations are well established in the algorithms community [8]). Further, we also present a variety of design optimizations that result in a practical performance which is well within the theoretical bounds.

In order to empirically validate its utility, we have evaluated the bouquet approach on PostgreSQL and a popular commercial DBMS. Our experiments utilize a rich set of complex decision support queries sourced from the TPC-H and TPC-DS benchmarks. The query workload includes selectivity spaces with as many as *five* error-prone dimensions, thereby capturing environments that are extremely challenging from a robustness perspective. Our performance results indicate that the bouquet approach typically provides *orders of magnitude* improvements, as compared to the optimizer's native choices. As a case in point, for Query 19 of the TPC-DS benchmark with 5 error prone join selectivities, the worst-case sub-optimality plummeted from about 10^6 to just 10! The potency of the approach is also indicated by the fact that for many queries, the bouquet's average performance is within 4 times of the corresponding PICs.

What is even more gratifying is that the above performance profiles are *conservative* since we assume that at every plan switch, *all* previous intermediate results are completely thrown away – in practice, it is conceivable that some of these prior results could be retained and reused in the execution of a future plan.

Apart from improving robustness, there is another major benefit of the bouquet mechanism: On a given database, the execution strategy for a particular query instance, i.e. the sequence of plan executions, is *repeatable* across different invocations of the query instance – this is in marked contrast to prior approaches wherein plan choices are influenced by the current state of the database statistics and the query construction. Such stability of performance is especially important for industrial applications, where considerable value is attributed to reproducible performance characteristics [3].

Finally, with regard to implementation, the bouquet technique can be largely constructed using techniques (e.g. abstract plan costing) that have already found expression in modern DB engines, as explained later in Section 5.5.

Thus far, we had tacitly assumed the optimizer's *cost model* to be perfect – that is, only *optimizer costs* were used in the evaluations. While this assumption is certainly not valid in practice, improving the model quality is, in principle, an orthogonal problem to that of estimation. Notwithstanding, we also analyze the robustness guarantees in the presence of bounded modeling errors. Moreover, to positively verify robustness improvements, explicit run-time evaluations are also included in our experimental study.

In closing, we wish to highlight that from a deployment perspective, the bouquet technique is in-

tended to *complementarily co-exist* with the classical optimizer setup, leaving it to the user or DBA to make the choice of which system to use for a specific query instance – essential factors that are likely to influence this choice are discussed in the epilogue.

Organization The remainder of the paper is organized as follows: In Section 2, a precise description of the robust execution problem is provided, along with the associated notations. Theoretical bounds on the robustness provided by the bouquet technique are presented in Section 3. We then discuss its design methodology, the compile-time aspects in Section 4 and the run-time mechanisms in Section 5. The experimental framework and performance results are reported in Section 6. Related work is reviewed in Section 7, while Section 8 presents a critical review of the bouquet approach. Finally, we conclude in Section 9.

2 Problem Framework

In this section, we present our robustness model, the associated performance metrics, and the notations used in the sequel. Robustness can be defined in many different ways and there is no universally accepted metric [12] – here, we use the notion of performance sub-optimality to characterize robustness.

The error-prone selectivity space is denoted as ESS and its dimensionality D is determined by the number of error-prone selectivity predicates in the query. The space is represented by a grid of D-dimensional points with each point $q(s_1, s_2, ..., s_D)$ corresponding to a unique query with selectivity s_j on the j^{th} dimension. The cost of a plan P_i at a query location q in ESS is denoted by $c_i(q)$.

For simplicity, we assume that the estimated query locations and the actual query locations are uniformly and independently distributed over the entire discretized selectivity space – that is, all estimates and errors are equally likely. This definition can easily be extended to the general case where the estimated and actual locations have idiosyncratic probability distributions.

Given a user query Q, denote the optimizer's *estimated* location of this query by q_e and the *actual* location at runtime by q_a. Next, denote the plan chosen by the optimizer at q_e as P_{oe}, and the optimal plan at q_a by P_{oa}. With these definitions, the sub-optimality incurred due to using P_{oe} at q_a is simply defined as the ratio:

$$SubOpt(q_e, q_a) = \frac{c_{oe}(q_a)}{c_{oa}(q_a)} \qquad \forall q_e, q_a \in ESS \tag{1}$$

with $SubOpt$ ranging over $[1, \infty)$. The worst-case $SubOpt$ for a given query location q_a is defined to be wrt the q_e that results in the maximum sub-optimality, that is, where selectivity inaccuracies have the maximum adverse performance impact:

$$SubOpt_{worst}(q_a) = \max_{q_e \in ESS}(SubOpt(q_e, q_a)) \quad \forall q_a \in ESS \tag{2}$$

With the above, the global worst-case is simply defined as the (q_e, q_a) combination that results in the maximum value of $SubOpt$ over the entire ESS, that is,

$$\textbf{MSO} = \max_{q_a \in ESS}(SubOpt_{worst}(q_a)) \tag{3}$$

Further, given the uniformity assumption about the distribution of estimated and actual locations, the average sub-optimality over ESS is defined as:

$$\textbf{ASO} = \frac{\sum_{q_e \in ESS} \sum_{q_a \in ESS} SubOpt(q_e, q_a)}{\sum_{q_e \in ESS} \sum_{q_a \in ESS} 1} \tag{4}$$

The above MSO and ASO definitions are appropriate for the way that modern optimizers behave, wherein selectivity estimates are made at compile-time, and a single plan is executed at run-time. However, in the plan bouquet technique, neither of these characteristics is true – error-prone selectivities are not estimated at compile-time, and multiple plans may be invoked at run-time. Notwithstanding, we can still compute the corresponding statistics by: (a) substituting q_e with a "don't care" $*$; (b) replacing P_{oe} with P_b to denote the plan bouquet mechanism; and (c) having the cost of the bouquet, $c_b(q_a)$, include the overheads incurred by the exploratory partial executions. Further, the running selectivity location, as progressively discovered by the bouquet mechanism, is denoted by q_{run}.

Even when the bouquet algorithm performs well on the MSO and ASO metrics, it is possible that for some specific locations $q_a \in \mathsf{ESS}$, it performs poorer than the worst performance of the native optimizer – it is therefore harmful for the queries associated with these locations. This possibility is captured using the following *MaxHarm* metric:

$$\mathbf{MH} = \max_{q_a \in ESS} \left(\frac{SubOpt(*, q_a)}{SubOpt_{worst}(q_a)} - 1 \right) \tag{5}$$

Note that MH values lie in the range $(-1, MSO_{bouquet} - 1]$ and harm occurs whenever MH is positive.

An assumption that fundamentally underlies the entire bouquet mechanism is that of *Plan Cost Monotonicity* (PCM) – that is, the costs of the POSP plans increase monotonically with increasing selectivity values. This assumption has often been made in the literature [5, 6, 14], and holds for virtually all the plans generated by PostgreSQL on the benchmark queries. The only exception we have found is for queries featuring *existential* operators, where the POSP plans may exhibit *decreasing* monotonicity with selectivity. Even in such scenarios, the basic bouquet technique can be utilized by the simple expedient of plotting the ESS with $(1 - s)$ instead of s on the selectivity axes. Thus, only queries whose optimal cost surfaces have a maxima or minima in the *interior* of the error space, are not amenable to our approach.

3 Robustness Bounds

We begin our presentation of the plan bouquet approach by characterizing its performance bounds with regard to the MSO metric, initially for the 1D scenario, and then extending it to the general multi-dimensional case.

3.1 1D Selectivity Space

As described in the Introduction, the 1D PIC curve is discretized by projecting a graded progression of isocost steps onto the curve. We assume that the PIC is an increasing function (by virtue of PCM) and continuous throughout ESS; its minimum and maximum costs are denoted by C_{min} and C_{max}, respectively. Now, specifically consider the case wherein the isocost steps are organized in a *geometric* progression with initial value a ($a > 0$) and common ratio r ($r > 1$), such that the PIC is sliced with $m = log_r\lceil \frac{C_{max}}{C_{min}} \rceil$ cuts, $IC_1, IC_2, \ldots IC_m$, satisfying the boundary conditions $a/r < C_{min} \leq IC_1$ and $IC_{m-1} < C_{max} = IC_m$, as shown in Figure 5.

For $1 \leq k \leq m$, denote the selectivity location where the k^{th} isocost step (IC_k) intersects the PIC by q_k and the corresponding bouquet plan as P_k. All the q_k locations are unique by definition due to the PCM and continuity requirements on the PIC curve. However, it is possible that some of the P_k plans may be common to multiple intersection points (e.g. in Figure 3, plan P1 was common to steps IC_1 through IC_4). Finally, for mathematical convenience, assign q_0 to be 0.

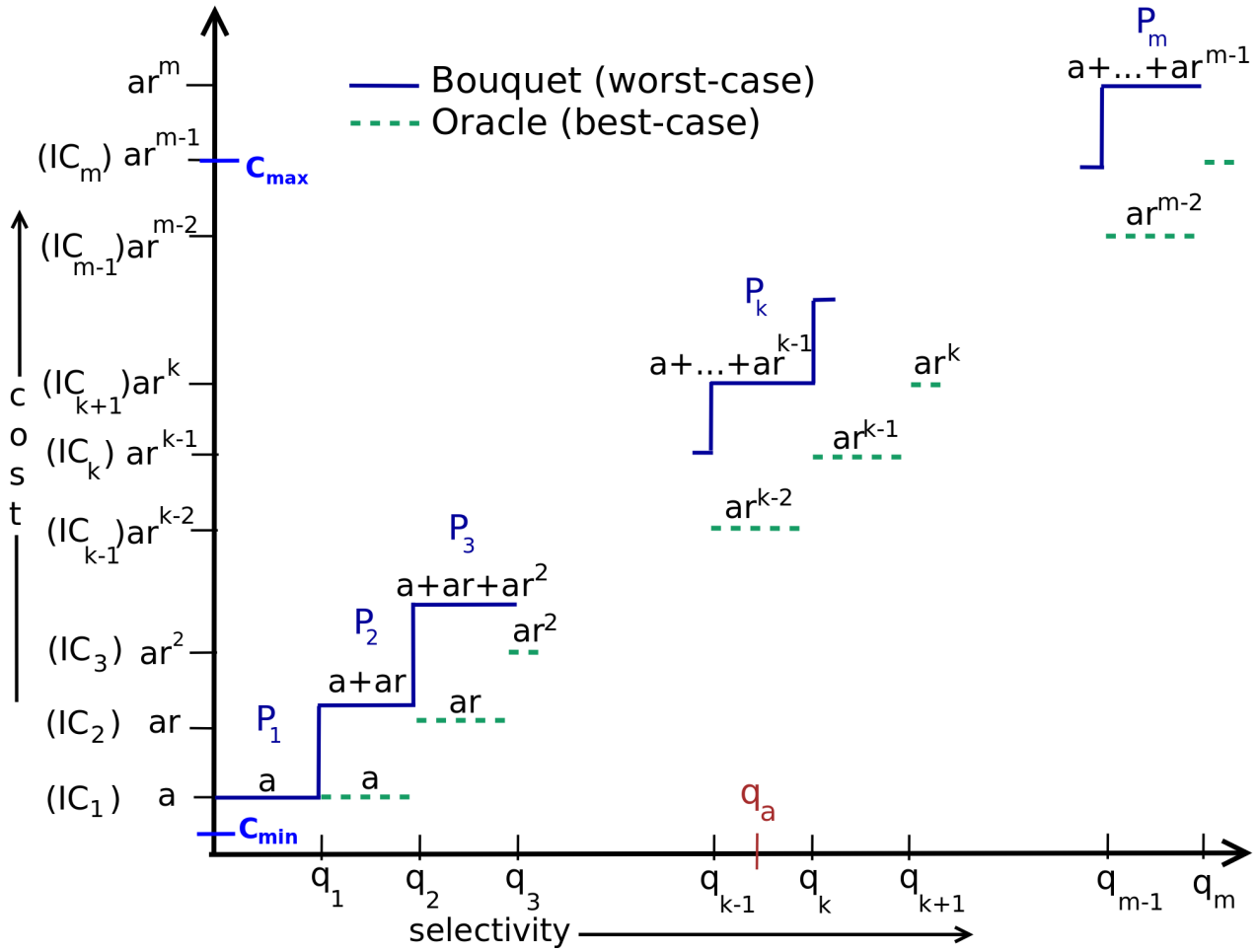

Figure 5: 1D Selectivity Space

With this framework, the bouquet execution algorithm operates as follows in the most general case, where a different plan is associated with each step: We start with plan P_1 and budget IC_1, progressively working our way up through the successive bouquet plans $P_2, P_3, \ldots$ until we reach the first plan P_k that is able to fully execute the query within its assigned budget IC_k. It is easy to see that the following lemma holds:

Lemma 1 *If q_a resides in the range $(q_{k-1}, q_k]$, $1 \leq k \leq m$, then plan P_k executes it to completion in the bouquet algorithm.*

Proof 1 We prove by contradiction: If q_a was located in the region $(q_k, q_{k+1}]$, then P_k could not have completed the query due to the PCM restriction. Conversely, if q_a was located in $(q_{k-2}, q_{k-1}]$, P_{k-1} itself would have successfully executed the query to completion. With similar reasoning, we can prove the same for the remaining regions that are beyond q_{k+1} or before q_{k-2}.

Performance Bounds Consider the generic case where q_a lies in the range $(q_{k-1}, q_k]$. Based on Lemma 1, the associated worst case cost of the bouquet execution algorithm is given by the following expression:

$$C_{bouquet}(q_a) = cost(IC_1) + cost(IC_2) + ... + cost(IC_k)$$

```
for step = 1 to m do
    start executing P_step
    while run_cost(P_step) ≤ cost-budget(IC_step) do
        execute P_step
        if P_step finishes execution then return query result
    stop P_step
```

Figure 6: Bouquet Algorithm (1D)

$$= a + ar + ar^2 + ... + ar^{k-1} = \frac{a(r^k - 1)}{r - 1} \tag{6}$$

The corresponding cost for an "oracle" algorithm that magically apriori knows the correct location of q_a is lower bounded by ar^{k-2}, due to the PCM restriction. Therefore, we have

$$SubOpt(*, q_a) \leq \frac{\frac{a(r^k-1)}{r-1}}{ar^{k-2}} = \frac{r^2}{r-1} - \frac{r^{2-k}}{r-1} \leq \frac{r^2}{r-1} \tag{7}$$

Note that the above expression is *independent* of k, and hence of the specific location of q_a. Therefore, we can state for the entire selectivity space, that:

Theorem 1 *Given a query* Q *on a 1D error-prone selectivity space, and the associated PIC discretized with a geometric progression having common ratio r, the bouquet execution algorithm ensures that:*

$$MSO \leq \frac{r^2}{r-1}$$

Further, the choice of r can be optimized to minimize this value – the RHS reaches its minima at $r = 2$, at which the value of MSO is **4**. The following theorem shows that this is the *best* performance achievable by any deterministic online algorithm – leading us to conclude that the *doubling* based discretization is the ideal solution.

Theorem 2 *No deterministic online algorithm can provide an MSO guarantee lower than 4 in the 1D scenario.*

Proof 2 We prove by contradiction, assuming there exists an optimal online robust algorithm, R* with a MSO of f, $f < 4$.

Firstly, note that R* must have a monotonically increasing sequence of plan execution costs, $a_1, a_2, \ldots, a_{k^*+1}$ in its quest to find a plan P_{k^*+1} that can execute the query to completion. The proof is simple: If $a_i > a_j$ with $i < j$, then we could construct another algorithm that skips the a_j execution and still execute the query to completion using P_{k^*+1}, and therefore has less cumulative overheads than R*, which is not possible by definition.

Secondly, if R* stops at P_{k^*+1}, then q_a has to necessarily lie in the range $(q_{k^*}, q_{k^*+1}]$ (Lemma 1 holds for any monotonic algorithm). Therefore, the worst-case performance of R* is given by $\frac{\sum_{i=1}^{i=k^*+1} a_i}{a_{k^*}} \leq f$. Since q_a could be chosen to lie in any interval, this inequality should hold true across all intervals, i.e.

$$\forall j \in 1, 2, \ldots, k^*: \quad \frac{\sum_{i=1}^{i=j+1} a_i}{a_j} \leq f$$

Using the notation A_j to represent $\sum_{i=1}^{j} a_i$ and Y_j to represent the ratio $\frac{A_{j+1}}{A_j}$, we can rewrite the above as:

$\frac{A_{j+1}}{a_j} \leq f \Rightarrow A_{j+1} \leq f(A_j - A_{j-1}) \Rightarrow \frac{A_{j+1}}{A_j} \leq f\frac{(A_j - A_{j-1})}{A_j}$

that is, $Y_j \leq f(1 - \frac{1}{Y_{j-1}})$.

We can show through elementary algebra that $\forall z > 0, \quad (1 - \frac{1}{z}) \leq \frac{z}{4}$. Therefore, we have that $Y_j \leq (\frac{f}{4})Y_{j-1}$, leading to $Y_{k^*} \leq (\frac{f}{4})^{k^*-1} Y_1$. Using the assumption of $f < 4$, we can find a sufficiently large k^* such that $(\frac{f}{4})^{k^*-1} Y_1 < 1$. Hence, $Y_{k^*} < 1$ which implies that $A_{k^*+1} < A_{k^*}$, a contradiction.

3.2 Multi-dimensional Selectivity Space

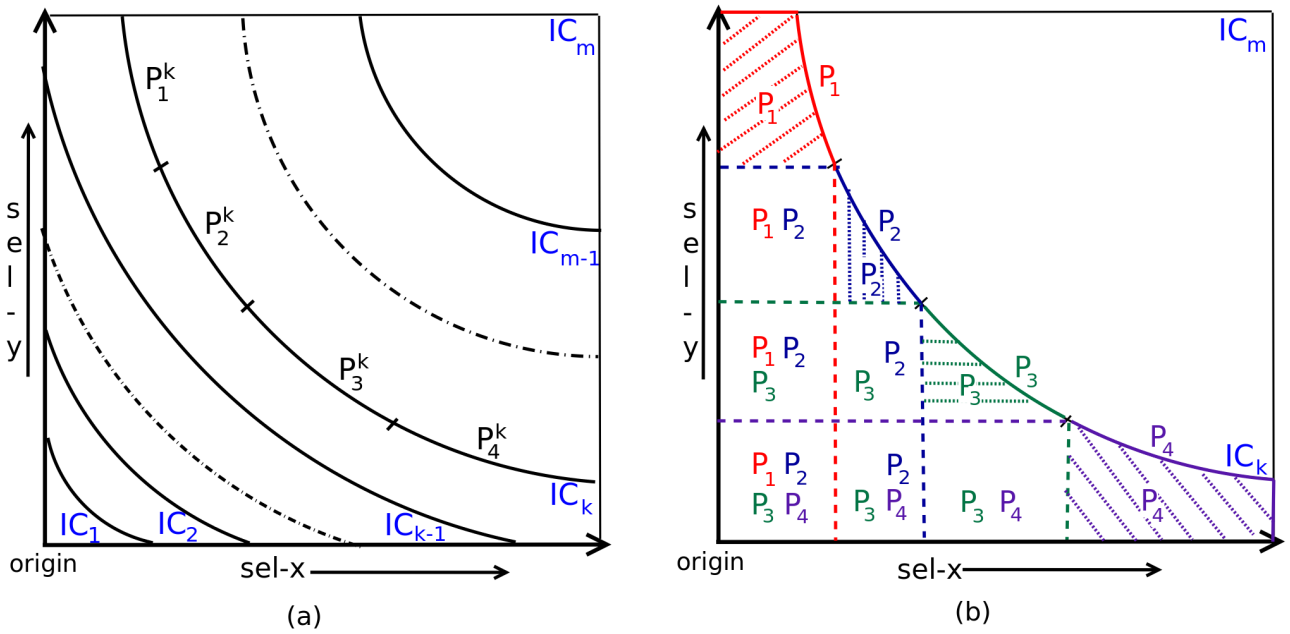

Figure 7: 2D Selectivity Space

We now move to the general case of multi-dimensional selectivity error spaces. A sample 2D scenario is shown in Figure 7a, wherein the isocost surfaces IC_k are represented by *contours* that represent a continuous sequence of selectivity locations (in contrast to the single location in the 1D case). Further, multiple bouquet plans may be present on each individual contour as shown for IC_k wherein four plans, P_1^k, P_2^k, P_3^k, P_4^k, are the optimizer's choices over disjoint x, y selectivity ranges on the contour. Now, to decide whether q_a lies below or beyond IC_k, in principle *every* plan on the IC_k contour has to be executed – only if none complete, do we know that the actual location definitely lies beyond the contour.

This need for exhaustive execution is highlighted in Figure 7b, where for the four plans lying on IC_k, the regions in the selectivity space on which each of these plans is guaranteed to complete within the IC_k budget are enumerated (the contour superscripts are omitted in the figure for visual clarity). Note that while several regions are "covered" by multiple plans, each plan also has a region that it alone covers – the hashed regions in Figure 7b. For queries located in such regions, only the execution of the associated unique plan would result in confirming that the query is within the contour.

The basic bouquet algorithm for the multi-dimensional case is shown in Figure 8, using the notation n_k to represent the number of plans on contour k.

```
for cid = 1 to m do                                          ▷ for each cost-contour cid
    for i = 1 to n_cid do                                    ▷ for each plan on cid
        start executing P_i^cid
        while running-cost(P_i^cid) ≤ cost-budget(IC_cid) do
            execute plan P_i^cid                             ▷ cost limited execution
            if P_i^cid finishes execution then
                return query result
        stop executing P_i^cid
```

Figure 8: Multi-dimensional Bouquet Algorithm

Performance Bounds Given a query Q with q_a located in the range $(IC_{k-1}, IC_k]$, the worst-case total execution cost for the multi-D bouquet algorithm is given by

$$C_{bouquet}(q_a) = \sum_{i=1}^{k} [n_i \times cost(IC_i)] \tag{8}$$

Using ρ to denote the number of plans on the *densest* contour, and upper-bounding the values of the n_i with ρ, we get the following performance guarantee:

$$C_{bouquet}(q_a) \leq \rho \times \sum_{i=1}^{k} cost(IC_i) \tag{9}$$

Now, following a similar derivation as for the 1D case, we arrive at the following theorem:

Theorem 3 *Given a query* Q *with a multidimensional error-prone selectivity space, the associated PIC discretized with a geometric progression having common ratio r and maximum contour plan density* ρ*, the bouquet execution algorithm ensures that:* $MSO \leq \rho \frac{r^2}{r-1}$

Proof 3 *Setting* $r = 2$ *in this expression ensures that* $MSO \leq 4\rho$.

3.3 Minimizing IsoCost Surface Plan Density

To the best of our knowledge, the above MSO bounds are the first such guarantees in the literature. While the 1D bounds are inherently strong giving a guarantee of 4 or better, the multi-dimensional bounds, however, depend on ρ, the maximum plan density over the isocost surfaces. Therefore, to have a practically useful bound, we need to ensure that the value of ρ is kept to the minimum.

This can be achieved through the *anorexic reduction* technique described in [14]. Here, POSP plans are allowed to "swallow" other plans, that is, occupy their regions in the ESS space, if the sub-optimality introduced due to these swallowings can be bounded to a user-defined threshold, λ. In [14], it was shown that even for complex OLAP queries, a λ value of 20% was typically sufficient to bring the number of POSP plans down to "anorexic levels", that is, a small absolute number within or around 10.

When we introduce the anorexic notion into the bouquet setup, it has two opposing impacts on the sub-optimality guarantees – on the one hand, the constant multiplication factor is increased by a factor $(1 + \lambda)$; on the other, the value of ρ is significantly reduced. Overall, the deterministic guarantee is altered from $4\rho_{\text{POSP}}$ to $4(1+\lambda)\rho_{\text{ANOREXIC}}$.

Empirical evidence that this tradeoff is very beneficial is shown in Table 1, which compares for a variety of multi-dimensional error spaces, the bounds (using Equation 8) under the original POSP

configuration and under an anorexic reduction ($\lambda = 20\%$). As a particularly compelling example, consider 5D_DS_Q19, a five-dimensional selectivity error space based on Q19 of TPC-DS – we observe here that the bound plunges by more than an order of magnitude, going down from 379 to 30.4.

Error Space	ρ POSP	MSO Bound	ρ ANOREXIC	MSO Bound
3D_H_Q5	11	**33**	3	**12.0**
3D_H_Q7	13	**34**	3	**9.6**
4D_H_Q8	88	**213**	7	**24.0**
5D_H_Q7	111	**342.5**	9	**37.2**
3D_DS_Q15	7	**23.5**	3	**12.0**
3D_DS_Q96	6	**22.5**	3	**13.0**
4D_DS_Q7	29	**83**	4	**17.8**
4D_DS_Q26	25	**76**	5	**19.8**
4D_DS_Q91	94	**240**	9	**35.3**
5D_DS_Q19	159	**379**	8	**30.4**

Table 1: Performance Guarantees (POSP versus Anorexic)

3.4 Cost Modeling Errors

Thus far, we had catered to arbitrary errors in selectivity estimation, but assumed that the cost model itself was perfect. In practice, this is certainly not the case, but if the modeling errors were to be unbounded, it appears hard to ensure robustness since, in principle, the estimated cost of any plan could be arbitrarily different to the actual cost encountered at run-time. However, we could think of an intermediate situation wherein the modeling errors are non-zero but *bounded* – specifically, the estimated cost of any plan, given correct selectivity inputs, is known to be within a δ error factor of the actual cost. That is, $\frac{c_{estimated}}{c_{actual}} \in [\frac{1}{(1+\delta)}, (1+\delta)]$.

Our construction is lent credence to by the recent work of [23], wherein static cost model tuning was explored in the context of PostgreSQL – they were able to achieve an average δ value of around 0.4 for the TPC-H suite of queries.

This "unbounded estimation errors, bounded modeling errors" framework is then amenable to robustness analysis and leads to following result.

Theorem 4 *If the cost-modeling errors are limited to error-factor δ with regard to the actual cost, the bouquet algorithm ensures that:* $MSO \leq (1+\delta)^2 \rho \frac{r^2}{r-1}$

Proof 4 Recall from Equation 9 that, for any given query instance $q_a \in (IC_{k-1}, IC_k]$, the performance of bouquet algorithm with perfect cost model assumption was given by:

$$C_{bouquet}(q_a) \leq \rho \times \sum_{i=1}^{k} cost(IC_i)$$

Now, in the presence of cost modeling error, the sub-optimality of the bouquet technique will degrade most when all the partial execution costs were underestimated and the corresponding cost for "oracle" algorithm is overestimated to the largest extent. The actual costs in such a case would be -

$$C_{bouquet}(q_a) \leq \rho \times \sum_{i=1}^{k} (1+\delta)\, cost(IC_i) \quad and \quad C_{oracle}(q_a) \geq \frac{cost(IC_{k-1})}{(1+\delta)}$$

Thus, the sub-optimality in this case would be:

$$SubOpt(*, q_a) = \frac{C_{bouquet}}{C_{oracle}} \leq \rho(1+\delta)^2 \frac{\sum_{i=1}^{k} cost(IC_i)}{cost(IC_{k-1})} = \rho(1+\delta)^2 \times \frac{1}{ar^{k-2}} \times \frac{a(r^k-1)}{r-1}$$

$$SubOpt(*, q_a) \leq \rho(1+\delta)^2 \frac{r^2}{r-1} \tag{10}$$

Thus, we can conclude from Theorem 3 and Theorem 4 that,

$$MSO_{bounded\_modeling\_error} \leq MSO_{perfect\_model} * (1+\delta)^2 \tag{11}$$

The effectiveness of this result is clear from the fact that, when $\delta = 0.4$, corresponding to the average in [23], the MSO increases by at most a factor of 2. Such low value of δ is also corroborated by the views of industry experts [24] based on their experience in real world scenarios.

4 Bouquet: Compile-Time

In this section, we describe the compile-time aspects of the bouquet algorithm, whose complete workflow is shown in Figure 9.

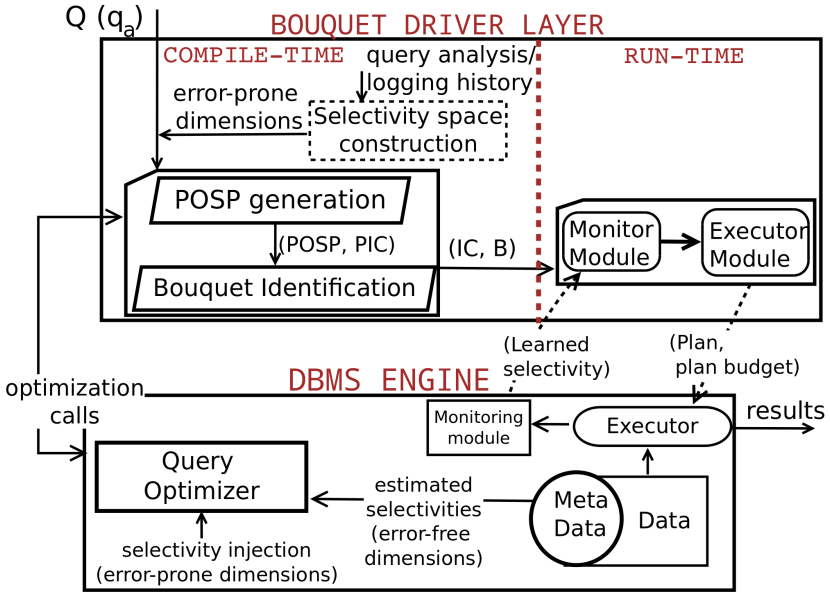

Figure 9: Architecture of Bouquet Mechanism

4.1 Selectivity Space Construction

Given a user query Q, the first step is to identify the error-prone selectivity dimensions in the query. For this purpose, we can leverage the approach proposed in [16], wherein a set of uncertainty modeling

rules are outlined to classify selectivity errors into categories ranging from "no uncertainty" to "very high uncertainty". Alternatively, a log could be maintained of the errors encountered by similar queries in the workload history. Finally, there is always the fallback option of making *all* predicates where selectivities are evaluated, to be selectivity dimensions for the query.

The chosen dimensions form the ESS selectivity space. In general, each dimension ranges over the entire [0,100] percentage range – however, due to schematic constraints, the range may be reduced. For instance, the maximum legal value for a PK-FK join is the reciprocal of the PK relation's minimum row cardinality.

4.2 POSP Generation

The next step is to determine the parametric optimal set of plans (POSP) over the entire ESS. Producing the complete POSP set requires repeated invocations of the query optimizer at a high degree of resolution over the space. This process can, in principle, be computationally very expensive, especially for higher-dimensional spaces. However, user queries are often submitted through "canned" form-based interfaces – for such environments it appears feasible to offline *precompute* the entire POSP set.

Further, even when this is not the case, the overheads can be made manageable by leveraging the following observation: The full POSP set is not required, only the subset that lies on the isocost surfaces. Therefore, we begin by optimizing the two locations at the corners of the principal diagonal of the selectivity space, giving us C_{min} and C_{max}. From these values, the costs of all the isocost contours are computed. Then, the ESS is divided into smaller hypercubes, recursively dividing those hypercubes through which one or more isocost contours pass – a contour passes through a hypercube if its cost is within the cost range established by the corners of the hypercube's principal diagonal. The recursion stops when we reach hypercubes whose sizes are small enough that it is cheap to explicitly optimize all points within them (As specified in Section 2, ESS has been discretized in the form of a high resolution grid). In essence, only a narrow "band" of locations around each contour is optimized.

Finally, note that the POSP generation process is "embarrassingly parallel" since each location in the ESS can be optimized independent of the others. Therefore, hardware resources in the form of multi-processor multi-core platforms can also be leveraged to bring the overheads down to practical levels.

Selectivity Injection As discussed above, we need to be able to systematically generate queries with the desired ESS selectivities. One option is to, for each new location, suitably modify the query constants and the data distributions, but this is clearly impractically cumbersome and time-consuming. We have therefore taken an alternative approach in our PostgreSQL implementation, wherein the optimizer is instrumented to directly support *injection* of selectivity values in the cost model computations. Interestingly, some commercial optimizer APIs already support such selectivity injections to a limited extent (e.g. IBM DB2 [26]).

4.3 Plan Bouquet Identification

Armed with knowledge of the plans on each of the isocost contour surfaces, which is usually in the several tens or hundreds of plans, the next step is to carry out a cost-based anorexic reduction [14] in order to bring the plan cardinality down to a manageable number. That is, we identify a smaller set of plans, such that each replaced location now has a new plan whose cost is within $(1+\lambda)$ times the optimal cost. We denote the set of plans on the surface of IC_k with B_k and the union of these sets of plans provides the final plan bouquet i.e. $B = \cup_{k=1}^{m} B_k$. Finally, the isocost surfaces (IC), annotated with

their updated costs (the original costs are inflated by $1 + \lambda$ to account for the anorexic reduction), and B, the set of bouquet plans, are passed to the run-time phase.

5 Bouquet: Run Time

In this section, we present the run-time aspects of the bouquet mechanism, as per the work-flow shown in Figure 9.

The basic bouquet algorithm (Figure 8) discovers the location of a query by sequentially executing the set of plans on each contour in a cost-limited manner until either one of them completes, or the plan set is exhausted, forcing a jump to the next contour. Note that in this process, no explicit monitoring of selectivities is required since the execution statuses serve as implicit indicators of whether we have reached q_a or not. However, as we will show next, consciously tracking selectivities can aid in substantively curtailing the discovery overheads. In particular, the tracking can help to (a) reduce the number of plan executions incurred in crossing contours; and (b) develop techniques for increasing the selectivity movement obtained through each cost-limited plan execution.

5.1 Reducing Contour Crossing Executions

In this optimization, during the processing of a contour, the location of q_{run} is *incrementally* updated after each (partial) plan execution to reflect the additional knowledge gained through the execution. An example learning sequence is shown in Figure 10 – here, the q_{run} known at the conclusion of IC_{k-1} is progressively updated via q^1_{run} and q^2_{run} to reach q^3_{run} on IC_k, with the corresponding plan execution sequence being P_1, P_4, P_3 (the contour superscripts are omitted for ease of exposition). The important point to observe here is that the contour crossing was accomplished *without executing* P_2.

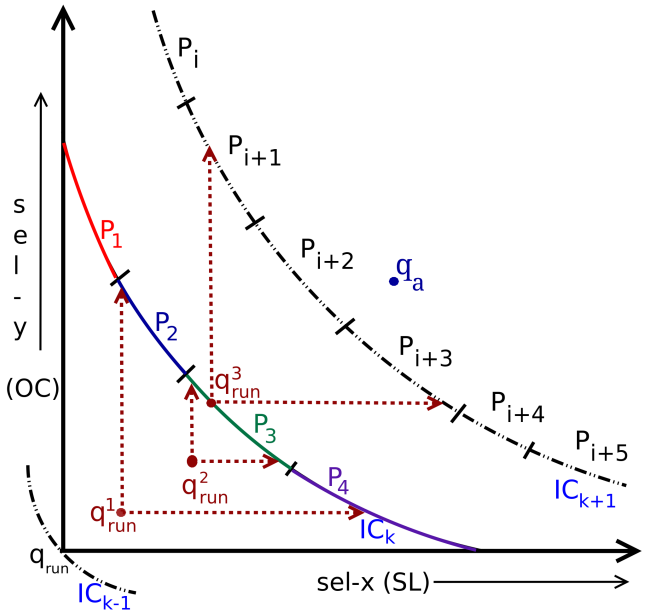

Figure 10: Minimizing Contour Crossing Executions

We now discuss how the plan execution sequence is decided. The strategy used is to ensure that at all times, the actual location is in the *first quadrant* with respect to the current location as origin – this invariant allows us to use the positive axes as a "pincer" movement towards reaching the desired

target, in the process eliminating from consideration some plans on the contour. Specifically, at each q_{run} location, we first identify *AxisPlans*, the set of bouquet plans present at the *intersection* of the isocost contour with the dimensional-axes corresponding to q_{run} as origin. For example, in Figure 10, *AxisPlans*(q_{run}) is comprised of P_4 and P_1, corresponding to the x and y dimensions, respectively. Then, from within this set, we heuristically pick, using a combination of structure and cost considerations, the plan that promises to provide the maximum movement towards q_a. The specific heuristic used is the following: The plans in *AxisPlans* are first ordered based on their costs at q_{run}, and then clustered into "equivalence groups" based on the closeness of these costs. From the cheapest equivalence group, the plan with an error-prone node occurring deepest in the plan-tree is chosen for execution. The expectation is that being cheapest at q_{run} provides the maximum spare budget, while having error-prone nodes deep within the plan-tree ensures that this spare budget is not uselessly spent on processing error-free nodes.

In Figure 10, the above heuristic happens to chose P_1 at q_{run} and thereby reach q^1_{run}. The process is repeated with q_{run} set to q^1_{run} – now *AxisPlans* (q^1_{run}) is $\{P_2, P_4\}$, and P_4 is chosen by the heuristic, resulting in a movement to q^2_{run}. Finally, with q_{run} set to q^2_{run}, *AxisPlans* (q^2_{run}) contains only P_3 which is executed to reach q^3_{run}, and hence IC_k. Note, as mentioned before, that P_2 is eliminated from consideration in this incremental process.

There is a further advantage of the incremental updates to q_{run}: When we reach q^3_{run} in Figure 10, we not only learn that q_a lies beyond IC_k but can also ab initio eliminate 3 plans (P_i, P_{i+4}, P_{i+5}) from the list of candidate plans for crossing IC_{k+1}, since these plans lie outside the first quadrant of q^3_{run}.

5.2 Monitoring Selectivity Movement

Having established the utility of incremental updates to q_{run}, we now go into the details of its implementation. Consider the scenario wherein Figure 10 represents the selectivity updation process for a TPC-H based query with error-prone join selectivities s_{SL} and s_{OC} on the x and y dimensions, respectively. Correspondingly, let plans P_1 through P_4 be as shown in Figure 11. Further, each node j of these plans is labeled with the corresponding tuple count, t_j, obtained at the end of the cost-limited execution – these annotations are explicitly shown for P_1 in Figure 11.

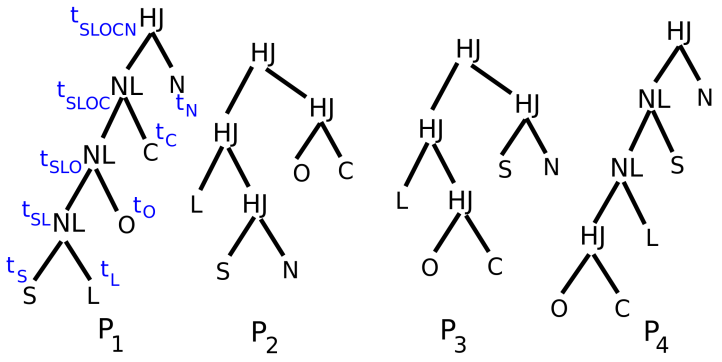

Figure 11: Plans on the k^{th} contour

After P_1's execution, the tuple count on node SL can be utilized to update the running selectivity $\hat{s}_{\text{SL}}$ as $\frac{t_{\text{SL}}}{|\text{S}|_e \times |\text{L}|_e}$ where $|\text{S}|_e$ and $|\text{L}|_e$ denote the cardinalities of the input relations to the SL join. The values in the denominator are clearly known before execution as these nodes are assumed to be error-free. Note that $\hat{s}_{\text{SL}}$ is a *lower bound* on s_{SL}, and therefore continues to maintain the "first quadrant" invariant required by the bouquet approach.

The other selectivity s_{oc}, is not present as an independent node in plan P_1. If we directly use $\hat{s}_{oc} = \frac{t_{sloc}}{t_{slo} \times |c|_e}$, there is a danger of *overestimation* wrt q_a(oc) since t_{slo} may not be known completely due to the cost-budgeted execution of P_1. Such overestimations may lead to violation of the "first quadrant" property, and are therefore impermissible. Consequently, we defer the updating of s_{oc} to the subsequent execution of plans P_4 and P_3 where it can be independently computed from fully known inputs.

In general, given any plan-tree, we can learn the lower bound for an error-prone selectivity only after the cardinalities of its inputs are completely known. This is possible when either the inputs are apriori error-free, or any error-prone inputs have been completely learnt through the earlier executions. The latter method of learning allows the bouquet approach to function even in the (unlikely) case where it does not possess plans with independent appearances for all the error-prone selectivities. In the above discussion, an implicit assumption is that all selectivities are *independent* with respect to each other – this is in conformance with the typical modeling framework of current optimizers. (Note that, the above discussion assumes *independence of selectivities* but not the other traditional assumptions like uniform distribution assumption or random placement assumption and hence there is a possibility of overestimation while learning any selectivity whenever complete inputs are not known).

5.3 Maximizing Selectivity Movement

Now we discuss how individual cost-limited plan executions can be modified to yield maximum movement of q_{run} towards q_a in return for the overheads incurred in their partial executions – that is, to "get the maximum selectivity bang for the execution buck".

In executing a budgeted plan to determine error-prone selectivities, we would ideally like the cost budget to be utilized as far as possible by the nodes in the plan operator tree that can provide us *useful* learning. However, there are two hurdles that come in the way: Firstly, the costs incurred by upstream nodes that *precede* the error nodes in the plan evaluation. Secondly, the costs incurred by the downstream nodes in the *pipeline* featuring the error nodes.

The first problem of upstream nodes can be ameliorated by preferentially choosing during the *AxisPlans* routine, as mentioned earlier, plans that feature the error-prone nodes deeper (i.e. earlier) in the plan-tree. The second problem of downstream nodes can be solved by deliberately breaking the pipeline immediately after first error node and *spilling* its output, which ensures that the downstream nodes do not get any data/tuples to process. These changes help to maximize the effort spent on executing the error-prone nodes, and thereby increase the selectivity movement with a given cost budget.

Movement Example We now illustrate, using the same example scenario as Figure 10, as to how spill-based execution is utilized to achieve increased selectivity movement. In Figure 12, the spilled versions of the plans P_1 through P_4 are shown, denoted using $\tilde{P}$. The modified selectivity discovery process using the spilled partial executions is shown in Figure 13, with the progressive selectivity locations being q^a_{run}, q^b_{run}, q^c_{run} and q^d_{run}.

The discovery process starts with executing plan $\tilde{P}_1$ until its cost-limit is reached. The tuple count on the error-prone node sL is then used to calculate $\hat{s}_{sL}$, as discussed earlier. Since the budget allotted for the full plan is now solely focused on learning s_{sL}, it is reasonable to expect that there will be materially more movement in s_{sL} as compared to executing generic P_1. In fact, it is easy to prove that, at the minimum, crossing of q_{run} from the third quadrant [1] f the P_1 segment to its fourth quadrant is *guaranteed* – this minimal case is shown in Figure 13 as location q^a_{run}.

[1]The quadrants for a curve (in 2D) are constructed by placing the negative X axis and the positive Y axis at the left-most point of the curve, and the positive X axis and the negative Y axis at the right-most point on the curve.

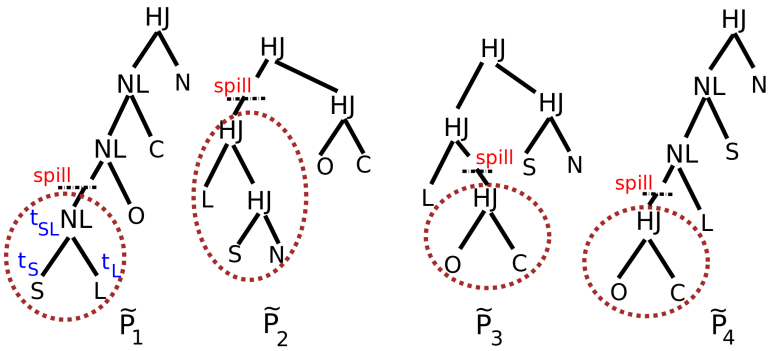

Figure 12: Plans (spilled version) and their movement direction

After $\tilde{P}_1$ exhausts its cost-budget, the *AxisPlans* routine chooses $\tilde{P}_4$ to take over, which starts learning s_{oc}, and ends up reaching at least q^b_{run} in Figure 13. Continuing in similar vein, $\tilde{P}_2$ is executed to reach q^c_{run}, and finally, $\tilde{P}_3$ is executed to reach q^d_{run} on the next contour. Due to focusing our energies on learning only a single selectivity in each plan execution, the movement of q_{run} follows a *Manhattan* profile from the origin upto q_a, as shown in Figure 13.

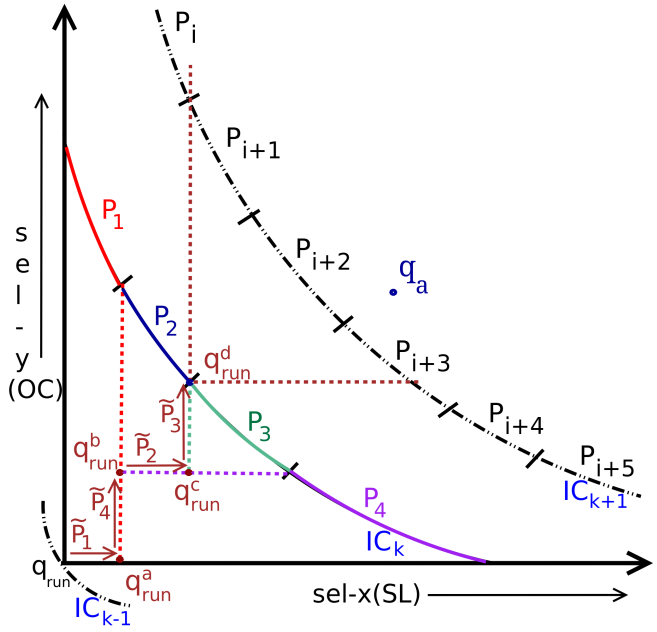

Figure 13: Maximizing Selectivity Movement

A high-level pseudocode of the full bouquet algorithm, incorporating the above optimizations, is presented in Figure 14.

Re-execution with Spilled Approach Although the spill-based approach serves to enhance the selectivity movement, there is also a downside – it requires a re-execution of the generic plan if the spilled plan's execution reaches the final destination on that dimension, signalled by the spilled plan completing before its budget is exhausted. This is because we need to verify whether the entire query, and not just the spilled segment, would complete within the cost budget. For instance, consider the case where q_a was located in the *fourth* quadrant of q^b_{run} – in this scenario, $\tilde{P}_4$ would complete early, after which

```
q_run = (0,0, ...,0);      cid = 1                                   ▷ initialization
loop
    P_cur = AxisPlanRoutine(q_run, cid)                              ▷ next plan selection
    while running-cost(P_cur) ≤ cost-budget(IC_cid) do
        execute P_cur                                                ▷ cost limited execution
        if P_cur finishes execution then
            return query result
    update q_run                                                     ▷ selectivity updation
    if optimal-cost(q_run) ≥ cost-budget(IC_cid) then
        cid ++                                                       ▷ early contour change
```

Figure 14: Optimized Bouquet algorithm

generic P_4 would have to be executed to deliver the query results.

Note that in order to maintain the MSO bound, we need to ensure that a contour is fully processed using *only* its assigned overall contour budget, even in the face of such re-executions. Therefore, we employ spilled executions *selectively* based on the following criteria: "Execute the spilled version only if the remaining number of dimensions to be learnt is less than the number of allotted plan executions remaining for the contour".

5.4 TPC-H Example

To make the above notions concrete, we now present an example of the bouquet algorithm operating on TPC-H Query 8. Assume that the error-prone selectivities of this query are the $P \bowtie L$, $S \bowtie L$ and $O \bowtie C$ joins, and that the actual selectivity location, q_a, is (0.8%, 0.25%, 30%). The optimal cost at this location is 2.35×10^5, and the worst case performance of the native optimizer, $SubOpt_{worst}(q_a)$, is 198.7.

The error-selectivity space of the query has 5 isocost contours lying between C_{min} = 1.95E4 and C_{max} = 5.5E5. Further, the number of POSP plans is 102, which reduce to just 9 plans after anorexic reduction with $\lambda = 20\%$. In this case, all 9 plans also feature in the bouquet, and the corresponding MSO bound (computed using Equation 8) is 18.5.

The detailed sequence of plan executions by the bouquet mechanism is shown in Table 2, where the selectivity learnt in each step is boldfaced. We also show, for each step, the plan employed, the assigned cost budget, and the overheads accumulated thus far. Further, the group of steps corresponding to individual contours are clubbed together in separate boxes.

In Table 2, we observe that the execution sequence is comprised of eight partial plan executions spanning four contours and five plans, and ending with the full execution of plan P_9 which returns the query results to the user. The overall MSO is **6.7**, a significant drop from the 198.7 of the native optimizer, and well within the bound of 18.5. Moreover, the overheads incurred in Table 2 are overstated – if optimizations such as sub-plan reuse are incorporated, the MSO would have gone down further to only **4**.

Had the above query been executed with the basic bouquet algorithm, it would have required 17 partial executions of 9 bouquet plans, resulting in an MSO of 10.7 – these statistics highlight the effectiveness of the optimizations described earlier in this section.

	PL	SL	OC	Plan	Cost-Budget	Overheads
0	0	0	0	-	-	-
1	**0.008**	0	0	$\widetilde{P}_4$	3.4×10^4	3.4×10^4
2	0.008	0	**1.0**	$\widetilde{P}_1$	6.8×10^4	1.02×10^4
3	0.008	**0.016**	1.0	$\widetilde{P}_8$	1.36×10^5	2.38×10^5
4	0.008	0.016	**4.0**	$\widetilde{P}_1$	1.36×10^5	3.74×10^5
5	**0.16**	0.016	4.0	$\widetilde{P}_4$	1.36×10^5	5.10×10^5
6	0.16	**0.25**	4.0	$\widetilde{P}_2$	2.72×10^5	7.82×10^5
7	0.16	0.25	**13.0**	$\widetilde{P}_1$	2.72×10^5	1.05×10^6
8	**0.8**	0.25	13.0	$\widetilde{P}_4$	2.72×10^5	1.32×10^6
9	0.8	0.25	**30.0**	P_9	2.72×10^5	1.58×10^6

Table 2: Example Bouquet Execution

5.5 Implementation Details

For implementing the bouquet mechanism, the database engine needs to support the following functionalities: (1) abstract plan costing; (2) selectivity injection during query optimization; (3) cost-limited partial execution of plans (generic and spilled); and (4) selectivity monitoring on a running basis. Abstract plan costing is supported by quite a few commercial engines including SQL Server [25], while limited selectivity injection is provided in DB2 [26]. The other two features were found to be easy to implement since they leverage pre-existing engine resources. For example, in PostgreSQL, the node-granularity tuple counter required for cost-limited execution, as well as selectivity monitoring, is available through the *instrumentation* data structure [29].

5.5.1 Cost-limited Execution in PostgreSQL

The basic bouquet approach requires, in principle, only a simple "timer" that keeps track of elapsed time and terminates plan executions if they exceed their assigned contour budgets. No material changes need to be made in the engine internals to support this feature (assuming perfect cost model).

To elaborate, we have an external program, the "Bouquet Driver" which treats the query optimizer and executor as black-boxes. First, it explores the ESS to determine the isocost contours and plan bouquet. Then it performs partial executions of plans using an *execution* client and a *monitoring* client. The execution client selects the plan to be executed next and the monitoring client keeps track of time elapsed and aborts the execution after the allotted time-budget has been exhausted. In PostgreSQL, this can be achieved by invoking the following command at the monitoring client: "*select pg_cancel_backend(process_id)*". The required *process_id* (of the execution client) can be found in the view *pg_stat_activity*, which is maintained by the engine itself.

5.6 Summary of Features

We complete this discussion of the mechanics of the bouquet approach with a synopsis of its distinctive features: (a) Compile-time estimation is completely eschewed for error-prone selectivities; (b) Plan switch decisions are triggered by predefined isocost contours (in contrast to dynamic criteria of [16, 17]); (c) Plan switch choices are restricted to an anorexic set of precomputed POSP plans; (d) AVI assumptions on intra-relational predicates are dispensed with since selectivities are explicitly

monitored; (e) A first-quadrant invariant between the actual selectivity and the running selectivity is maintained, supporting monotonic progress towards the objective.

6 Experimental Evaluation

We now turn our attention towards profiling the performance of the bouquet approach on a variety of complex OLAP queries, using the MSO, ASO and MH metrics enumerated in Section 2. In addition, we also provide experiments that show – (a) spatial distribution of robustness in ESS; (b) low bouquet cardinalities; (c) run time improvements using actual query executions; (d) scalability of the approach with large datasets; (e) low sensitivity of the MSO bound to λ parameter; and (e) that improvements extend to commercial databases as well.

Before going to the evaluation details, we start with the database and system setup used in evaluation and rationale behind choice of comparative techniques followed by a brief discussion on the compile-time overheads incurred by the bouquet algorithm.

6.1 Experimental Setup

Database Environment The test queries, whose full-text are given in the appendix, are chosen from the TPC-H and TPC-DS benchmarks to cover a spectrum of join-graph geometries, including *chain*, *star*, *branch*, etc. with the number of base relations ranging from 4 to 8. The number of error-prone selectivities range from 3 to 5 in these queries, all corresponding to join-selectivity errors, for making challenging multi-dimensional ESS spaces. We experiment with the TPC-H and TPC-DS databases at their default sizes of 1GB and 100GB, respectively, as well as larger scaled versions. Finally, the physical schema has indexes on all columns featuring in the queries, thereby maximizing the cost gradient $\frac{C_{max}}{C_{min}}$ and creating "hard-nut" environments for achieving robustness.

The summary query workload specifications are given in Table 3 – the naming nomenclature for the queries is xD_y_Qz, where x specifies the number of dimensions, y the benchmark (H or DS), and z the query number in the benchmark. So, for example, 3D_H_Q5 indicates a three-dimensional error selectivity space on Query 5 of the TPC-H benchmark.

Query	Join-graph (# relations)	$\frac{C_{max}}{C_{min}}$	Query	Join-graph (# relations)	$\frac{C_{max}}{C_{min}}$
3D_H_Q5	chain(6)	16	3D_DS_Q96	star(4)	185
3D_H_Q7	chain(6)	5	4D_DS_Q7	star(5)	283
4D_H_Q8	branch(8)	28	5D_DS_Q19	branch(6)	183
5D_H_Q7	chain(6)	50	4D_DS_Q26	star(5)	341
3D_DS_Q15	chain(4)	668	4D_DS_Q91	branch(7)	149

Table 3: Query workload specifications

System Environment For the most part, the database engine used in our experiments is a modified version of PostgreSQL 8.4 [28], incorporating the changes outlined in Section 5.5. We also present sample results from a popular commercial optimizer. The hardware platform is a vanilla Sun Ultra 24 workstation with 8 GB memory and 1.2 TB of hard disk.

In the remainder of this section, we compare the bouquet algorithm (with anorexic parameter $\lambda = 20\%$) against the native PostgreSQL optimizer, and the SEER robust plan selection algorithm [13]. SEER uses a mathematical model of plan cost behavior in conjunction with anorexic reduction to

provide replacement plans that, at all locations in ESS, either improve on the native optimizer's performance, or are worse by at most the λ factor – it is therefore expected to perform better than the native optimizer on our metrics. It is important to note here that, in the SEER framework, the comparative yardstick is P_{oe}, the optimal plan at the *estimated* location, whereas in our work, the comparison is with P_{oa}, the optimal plan at the *actual* location.[2]

For ease of exposition, we will hereafter refer to the bouquet algorithm, the native optimizer, and the SEER algorithm as **BOU**, **NAT** and **SEER**, respectively, in presenting the results.

6.2 Rationale behind choice of comparative technique

We have chosen to compare the bouquet technique with SEER and not to compare the performance with re-optimization techniques. The detailed reasoning behind these choices is given below.

6.2.1 Choice of SEER

The bouquet technique provides guarantees on MSO across all error-combinations in ESS i.e. irrespective of q_e and q_a, it provides an absolute bound on $\frac{technique\_cost(q_a)}{optimal\_cost(q_a)}$. To the best of our knowledge, none of the already proposed techniques provided such absolute bound.

Although, SEER also did not provide an absolute bound on $\frac{technique\_cost(q_a)}{optimal\_cost(q_a)}$, but it guarantees that

$$\frac{technique\_cost(q_e, q_a)}{native\_cost(q_e, q_a)} \leq (1 + \lambda)$$

OR

$$\frac{technique\_cost(q_e, q_a)}{optimal\_cost(q_a)} \leq (1 + \lambda)\frac{native\_cost(q_e, q_a)}{optimal\_cost(q_a)}$$

That is,

$$MSO_{SEER} \leq (1 + \lambda)MSO_{native}$$

Also, the experimental analysis showed [13] that for a significant fraction of error situations (q_e, q_a) of the ESS, SEER provided significant help, which means

$$\frac{technique\_cost(q_e, q_a)}{native\_cost(q_e, q_a)} \ll 1$$

$$\Rightarrow technique\_cost(q_e, q_a) \ll native\_cost(q_e, q_a)$$

Hence, it can be expected that even if SEER cannot give an absolute bound, MSO_{SEER} is expected to be much less than MSO_{native}. Similar argument holds for ASO – improvement in many individual error combinations means improvement in ASO.

6.2.2 Re-optimization techniques

In this section, we use example query EQ (from Section 1) with $q_e = 70\%$ and show the performance of re-optimization techniques over all possible errors (Figure 15). It shows that the re-optimization techniques can have arbitrarily high MSO, even in the case of one error-prone selectivity. The performance of re-optimization techniques is expected to get worse in case of multiple selectivity errors

[2]Purely heuristic-based reoptimization techniques, such as POP [17] and Rio [4], are not included in the evaluation suite since their performance could be arbitrarily poor with regard to both P_{oe} and P_{oa}, as explained in the Section 6.2.

due to various reasons. Firstly, the heuristics that they employ are relatively more suited for 1D-spaces (e.g. near-optimal at principal diagonal corners imply near-optimal in interior space, where to introduce checkpoints, the approximate validity range by comparing only with structure equivalent plans). Secondly, the size of error space increases exponentially with dimensions and the effort required to recover from larger selectivity errors is expected to increase with the number of errors.

Performance of POP The execution will start with optimizer choice plan at the estimated location i.e. P5 with its validity range associated to it. But, in this case there is no other structure-equivalent (join-order without regard to commutativity) plan that is better than P5 in any range of selectivity. Hence, there will be no re-optimization and P5 will be executed to completeness irrespective of the actual selectivity observed at run-time. Now, since P5 has sub-optimality as large as 92.5, implying $MSO \geq 92.5$.

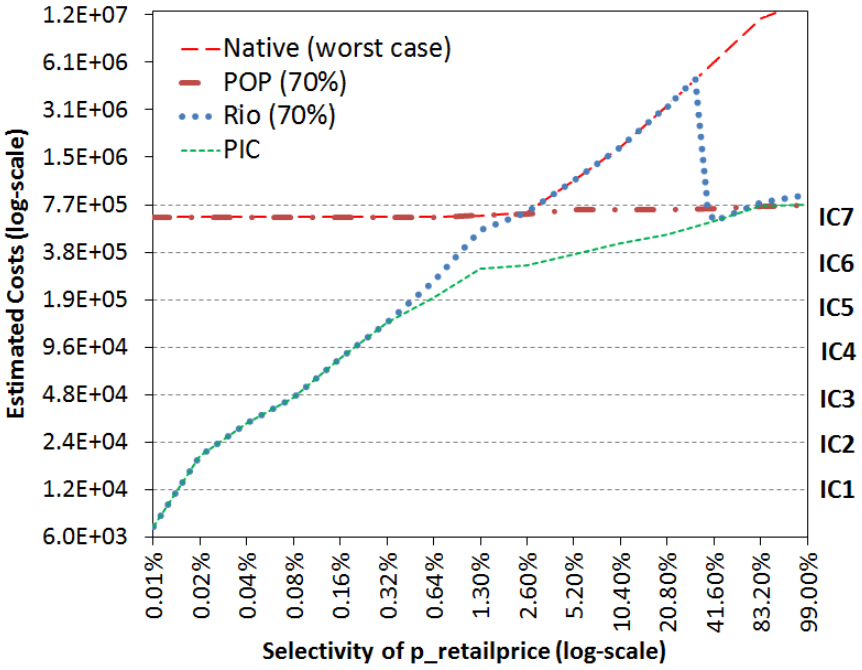

Figure 15: Performance of Re-optimization strategies with $q_e = 70\%$

Performance of Rio When the bounding box is limited around q_e (using $\triangle_- = 0.1$ and $\triangle_+ = 0.2$ [4]), it would cause P5 to be chosen as the robust choice plan inside the bounding box. In case the random sample obtained during execution identify the location to be outside the bounding box, it will cause re-optimization. In this case, the performance curve would be similar to that of POP due to fixed high initial overheads of plan P5.

Now, let us consider the case when the bounding box is assumed to cover full range of selectivity, i.e. $q_{min} \approx 0\%$, $q_e = 70\%$ and $q_{high} \approx 100\%$. Since these locations does not have either same optimal plan or one near-optimal plan in the set P1 and P5. Clearly, Rio tries to create switchable plan [4], such that one of the member plan is near optimal at each of the 3 locations. Fortunately enough, P4 is switchable with P1 (optimal at q_{min}) and near optimal at both q_e and q_{high}. So, Rio chooses a switchable plan with P1 and P4 as member plans. Now, for any particular q_a,

P1 is chosen if $q_{min} <= \hat{s}_{random} <= 35\%$ and P4 is chosen if $35\% < \hat{s}_{random} <= q_{high}$ causing MSO to be at least 9.7 (Figure 15).

Observations It is clear from above discussion that the re-optimization strategies make no visible effort to limit worst case performance and hence may cause very high MSO. In the current example, POP got stuck with a plan due to its validity range defined using structure equivalent plans only whereas Rio performed bad due to its heuristic assumption of robustness using only plans at principal corners of the bounding box and heuristic nature of switching decision.

6.3 Compile-time Overheads

The computationally expensive aspect of BOU's compile-time phase is the identification of the POSP set of plans in ESS. For this task, we use the contour-focused approach described in Section 4, which ignores most of the space lying between contours. In all of our queries, the number of contours was no more than 10. Therefore, the contour-POSP was generated within a *few hours* even for 5D scenarios on our generic workstation, which appears a feasible investment for canned queries. Moreover, as described in Section 4.2, these overheads could be brought down to a few minutes, thanks to the inherent parallelism in the task.

6.4 Worst-case Performance (MSO)

In Figure 16, the MSO performance is profiled, on a log scale, for a set of 10 representative queries submitted to NAT, SEER and BOU. The first point to note is that NAT is *not* inherently robust – to the contrary, its MSO is huge, ranging from around 10^3 to 10^7. Secondly, SEER also does not provide any material improvement on NAT – this may seem paradoxical at first glance, but is only to be expected once we realize that not *all* the highly sub-optimal (q_e, q_a) combinations in NAT were necessarily helped in the SEER framework. Finally, and in marked contrast, BOU provides *orders of magnitude* improvements over NAT and SEER – as a case in point, for 5D_DS_Q19, BOU drives MSO down from 10^6 to around just 10. In fact, even in absolute terms, it consistently provides an MSO of *less than ten* across all the queries.

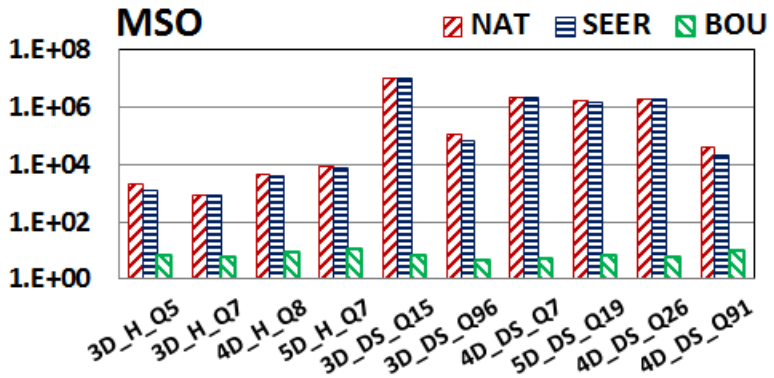

Figure 16: MSO Performance (log-scale)

6.5 Average-case Performance (ASO)

At first glance, it may be surmised that BOU's dramatic improvement in worst-case behavior is purchased through a corresponding deterioration of average-case performance. To quantitatively demonstrate that this is not so, we evaluate ASO for NAT, SEER and BOU in Figure 17, again on a log scale.

We see here that for some queries (e.g. 3D_DS_Q15), ASO of BOU is much better than that of NAT, while for the remainder (e.g. 4D_H_Q8) the performance is comparable. Even more gratifyingly, the ASO in absolute terms is typically less than 4 for BOU. On the other hand, SEER's performance is again similar to that of NAT – this is an outcome of the high dimensionality of the error space which makes it extremely difficult to find universally safe replacements that are also substantively beneficial.

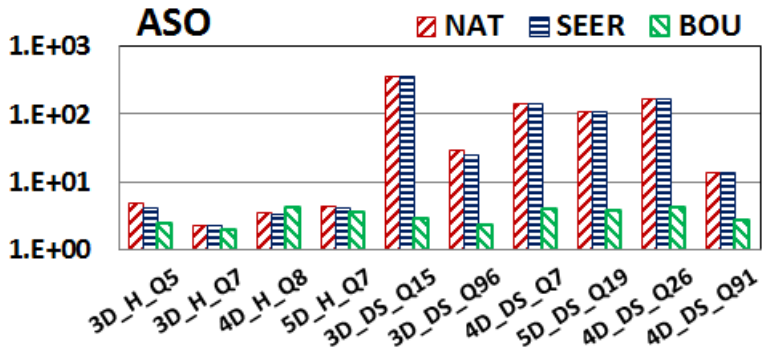

Figure 17: ASO Performance (log-scale)

6.6 Spatial Distribution of Robustness

We now profile for a sample query, namely 5D_DS_Q19, the percentage of locations for which BOU has a specific range of improvement over NAT. That is, the *spatial distribution* of enhanced robustness, $\frac{SubOpt_{worst}(q_a)}{SubOpt(*, q_a)}$. This statistic is shown in Figure 18, where we find that for the vast majority of locations (close to 90%), BOU provides *two or more orders of magnitude improvement* with respect to NAT. SEER, on the other hand, provides significant improvement over NAT for specific (q_e, q_a) combinations, but may not materially help the *worst-case* instance for each q_a. Therefore, we find that its robustness enhancement is less than 10 at all locations in the ESS.

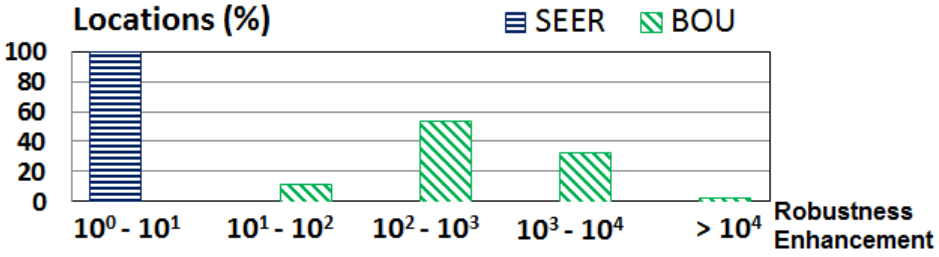

Figure 18: Distribution of enhanced Robustness (5D_DS_Q19)

6.7 Adverse Impact of Bouquet (MH)

Thus far, we have presented the improvements due to BOU. However, as highlighted in Section 2, there may be *individual* q_a locations where BOU performs poorer than NAT's worst-case, i.e. $SubOpt(*, q_a) > SubOpt_{worst}(q_a)$. This aspect is quantified in Figure 19 where the maximum harm is shown (on a linear scale) for our query test suite. We observe that BOU may be upto a factor of 4 worse than NAT. Moreover, SEER steals a march over BOU since it *guarantees* that MH never exceeds λ (= 0.2).

However, the important point to note is that the percentage of locations for which harm is incurred by BOU is less than 1% of the space. Therefore, from an overall perspective, the likelihood of BOU adversely impacting performance is rare, and even in these few cases the harm is limited (≤ MSO-1), especially when viewed against the order of magnitude improvements achieved in the beneficial scenarios.

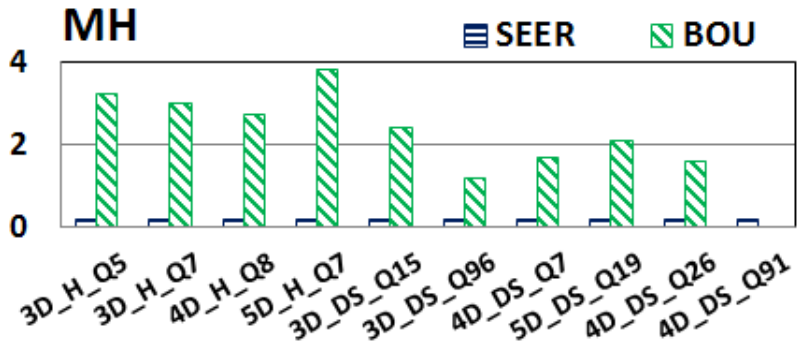

Figure 19: MaxHarm performance

6.8 Plan Cardinalities

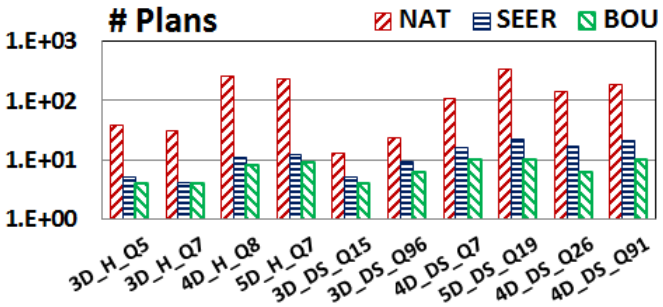

Figure 20: Plan Cardinalities (log-scale)

The plan cardinalities of NAT, SEER and BOU are shown on a log-scale in Figure 20. We observe here that although the original POSP cardinality may be in the several tens or hundreds, the number of plans in SEER is orders of magnitude lower, and those retained in BOU is even smaller – only around 10 or fewer, even for the 5D queries. This is primarily due to the initial anorexic reduction and the subsequent confinement to plan contours. The important implication of these statistics is that the bouquet size is, to the first degree of approximation, effectively *independent of the dimensionality and complexity of the error space*.

6.9 Query Execution Times (TPC-H)

To verify that the promised benefits of BOU are actually delivered at run-time, we also carried out experiments wherein query response times were explicitly measured for NAT and BOU. For this purpose, we crafted query instance 2D_H_Q8a, shown in Figure 21, whose q_a was $(33.7\%, 45.6\%)$, but NAT erroneously estimated the location to be $q_e = (3.8\%, 0.02\%)$ due to incorrect AVI assumptions.[3] As a

[3]We explicitly verified that there were no estimation errors in the remaining selectivity dimensions of the query.

```
select
    o_year,
    sum(case when nation = 'BRAZIL' then volume
    else 0 end) / sum(volume) as mkt_share
from
    (
    select
        DATE_PART('YEAR',o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) as volume,
        n2.n_name as nation
    from
        part, supplier, lineitem, orders,
        customer, nation n1, nation n2, region
    where
        p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = 'AMERICA'
        and s_nationkey = n2.n_nationkey
        and l_receiptdate ≤ l_shipdate + integer '60'
        and l_receiptdate ≤ l_commitdate + integer '130'
        and l_extendedprice ≤ 25000
        and c_name like '%er#000%'
        and c_acctbal ≤ 4000
    ) as all_nations
group by
    o_year
order by
    o_year
```

Figure 21: Example query based on TPC-H query 8

result, the plan chosen by NAT took almost *580 seconds* to complete, whereas the optimal plan at q_a finished in merely 16 seconds, i.e SubOpt(q_e, q_a) $\approx$ 36.

When BOU was invoked on the same 2D_H_Q8a query, it identified 6 bouquet plans spread across 7 isocost contours, resulting in an MSO bound of less than 20 (Equation 8). Subsequently, basic BOU produced the query result in about 117 seconds, involving 18 partial executions to cross 5 contours before the final full execution. Moreover, optimized BOU further brought the running time down to less than 70 seconds, using only 11 partial executions.

The isocost-contour-wise breakups of both basic and optimized BOU are given in Table 4, along with a comparative summary of their performance. Overall, the sub-optimality of optimized BOU is $\approx$ 4, almost an order of magnitude lower than that of NAT ($\approx$ 36). Note that the intended doubling of execution times across contours does not fully hold in Table 4 – this is an artifact of the imperfections in the underlying cost model of the PostgreSQL optimizer, compounded by our not having tuned this default model.

Contour ID	Avg Plan Exec. Time (in sec)	# Exec. (Basic BOU)	Time(sec) (Basic BOU)	# Exec. (Opt. BOU)	Time(sec) (Opt. BOU)
1	0.6	2	1.2	2	1.2
2	3.1	4	12.4	2	6.2
3	4.8	4	19.2	3	14.4
4	6.2	5	31.0	3	18.6
5	12.2	3	36.6	1	12.2
6	16.1	1	16.1	1	16.1
Total		**19**	**116.5**	**12**	**68.7**

Performance Summary	NAT	Basic BOU	Opt. BOU	Optimal
(in seconds)	579.4	116.5	68.7	16.1

Table 4: Bouquet execution for 2D_H_Q8a

6.10 Query Execution Times (for progressive errors)

The previous section shows an example TPC-H query where the selectivity estimation errors are successfully handled by bouquet technique. Now, we present evaluation of BOU technique under more challenging conditions, that are realized by – (1) forcing BOU to learn few more error-free selectivities in addition to the error-prone selectivities, (2) artificially creating progressively larger estimation errors.

Consider the 4D_H_Q8^b query (full text given in the appendix) with $q_a = (99\%, 7.5\%, 7.5\%, 99\%)$, which happens to be estimated by NAT as $q_e = (99\%, 0.5\%, 0.5\%, 99\%)$ (i.e. it is accurate in two dimensions and erroneous in the remaining two). In this scenario, the plan chosen by NAT took almost 500 seconds to complete, whereas the optimal plan at q_a finished in just 12 seconds.

Turning our attention to BOU, although two selectivities are accurate, we deliberately made it a difficult case by assuming that *all four* selectivities are erroneous. This forced BOU to learn all of them from scratch. In spite of this handicap, the basic BOU produced the query result in about 192 seconds, involving 41 partial executions before the final full execution. Moreover, the optimized BOU brought the running time down to around 48 seconds, using only 10 partial executions.

The iso-contour-wise breakups of both basic and optimized BOU are given in Table 5. Overall, the sub-optimality of optimized BOU is ≈ 4, an order of magnitude lower than that of NAT (≈ 42). Note that the intended doubling of execution times across contours does not fully hold in Table 5. This is an artifact of the imperfections in the underlying cost model of the optimizer, compounded by our not having tuned the default model.

The above experiment showcased an individual error instance. For the same setup, we also evaluated NAT and BOU on a *sequence* of locations located on the principal diagonal of the erroneous dimensions – that is, we gradually ramped up the estimation error from no error ($q_a = q_e$) to gross errors ($q_a \gg q_e$) on the two error-prone dimensions. The results of this experiment are shown in Figure 22, where the error locations are on the X-axis and the execution times (log-scale) are on the Y-axis.

We see in the figure that the SubOpt for NAT steadily increases with growing error and at the farthest location of the ESS (99%,99% 99%,99%), reaches as high as 2100. In contrast, the SubOpt for the basic BOU increases in the immediate neighborhood of the estimated location to around 17 at (99%, 7.5%, 7.5%, 99%) but then flattens out and remains relatively constant. At first glance, the flattening out might seem surprising given our claim that each new contour incurs twice the cost of the previous contour – the reason for this behavior is that, in this example, there were only 6 contours and 5 of them were already crossed while reaching $q_a = (99\%, 7.5\%, 7.5\%, 99\%)$ from origin, as shown in Table 5.

Contour ID	Avg Plan Exec. Time (in sec)	# Exec. (Basic BOU)	Time(sec) (Basic BOU)	# Exec. (Opt. BOU)	Time(sec) (Opt. BOU)
1	0.3	8	0.24	4	0.12
2	1.2	8	9.6	2	2.4
3	3.8	9	34.2	1	3.8
4	6.8	10	68.0	1	6.8
5	11.0	6	66.0	2	22.0
6	12.0	1	12.0	1	12.0
Total		42	192.2	11	48.2

Table 5: Bouquet execution for 4D_H_Q8

When we consider the optimized BOU, its qualitative profile is similar to that of basic BOU, but quantitatively, it brings down the maximum sub-optimality down from 17 to 4.2.

On the other hand, with regard to the harm metric, the maximum harm occurs for q_a = (99%, 0.5%, 0.5%, 99%). Here, basic BOU has a harm of 3.9 which is brought down to just 0.5 by the optimized BOU. Recall that, MH is defined with respect to $Subopt_{worst}(q_a)$, whose value here is 2.2.

Overall, if we restrict our attention to just this set of error locations, the usage of optimized BOU improves the MSO from 2100 to 4.2, ASO from 87.3 to 3.9 with MH value limited to just 0.5.

It is interesting to note that it takes only a small absolute error of around 8% (i.e. q_a = (99%, 8%, 8%, 99%), for NAT to start performing worse than basic BOU, and an even smaller error of around 1% (i.e. q_a = (99%, 1.5%, 1.5%, 99%), to perform worse than optimized BOU.

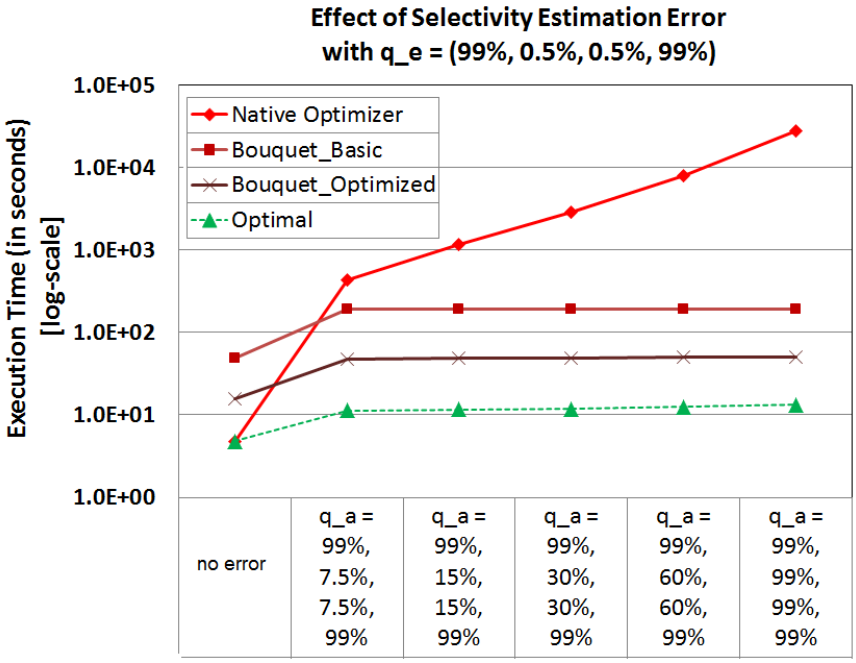

Figure 22: Execution Times (log-scale) with Progressive Errors

6.11 Scalability with Database Size

To study the impact of increase in database size on BOU's performance, we present the performance for the 3D_H_Q5 and 4D_H_Q8 queries on a scaled 10 GB TPC-H database in Figure 23. We find here, as should be expected, that the C_{max} value increases, resulting in a steeper $\frac{C_{max}}{C_{min}}$ gradient – for example, for 4D_H_Q8, the ratio increased from 28 to 40. This results in a significant increase in the MSO and ASO metrics for the native optimizer. However, BOU is largely unaffected with regard to both the guaranteed bound (bouquet cardinality) as well as the empirical performance.

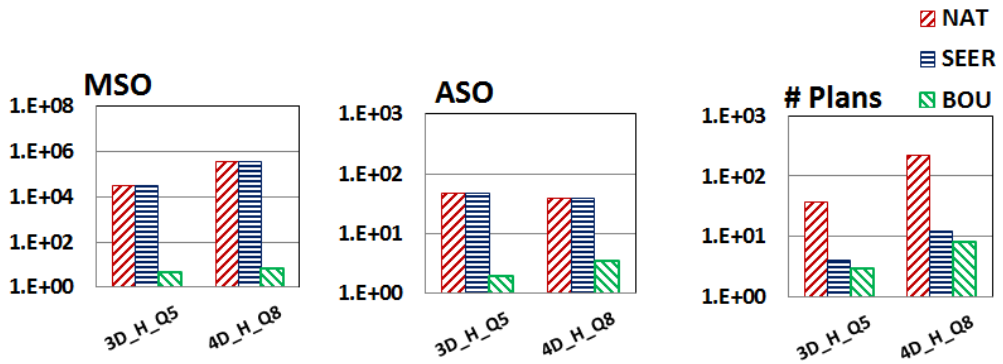

Figure 23: Performance (log-scale) with 10GB TPCH Database

6.12 MSO sensitivity to λ setting

The results thus far were all obtained with λ set to 20%, a value that had been found in [14] to routinely provide anorexic reduction over a wide range of database environments. However, a legitimate question remains as to whether the ideal choice of λ requires query and/or data-specific tuning. To assess this quantitatively, we show in Figure 24 the MSO bound values as a function of λ over the (0,100) percent range for a spectrum of query templates. The observation here is that the MSO bounds drop steeply as λ is increased to 10%, and subsequently are relatively flat in the (10,30) percent interval, suggesting that our 20% choice for λ is a safe bet in general.

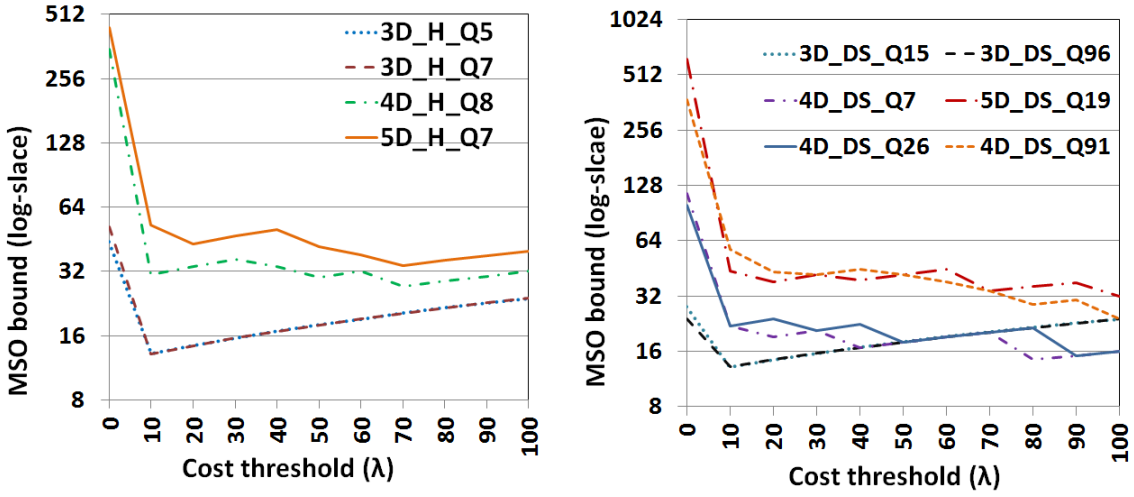

Figure 24: MSO bound vs Cost-threshold(λ)

6.13 Commercial Database Engine

All the results presented thus far were obtained on our instrumented PostgreSQL engine. We now present sample evaluations on a popular commercial engine, hereafter referred to as COM. Since COM's API does not directly support injection of selectivities, we constructed queries 3D_H_Q5b and 4D_H_Q8b(query text in appendix), wherein all error dimensions correspond to selection predicates on the base relations – the selectivities on such dimensions can be indirectly set up through changing only the constants in the query. The database and system environment remained identical to that of the PostgreSQL experiments.

Focusing on the performance aspects, shown in Figure 25, we find that here also large values of MSO and ASO are obtained for NAT and SEER. Further, BOU continues to provide substantial improvements on these metrics with a small sized bouquet. Again, the robustness enhancement is at least an order of magnitude for more than 90% of the query locations, without incurring any harm at the remaining locations (MH < 0). These results imply that our earlier observations are not artifacts of a specific engine.

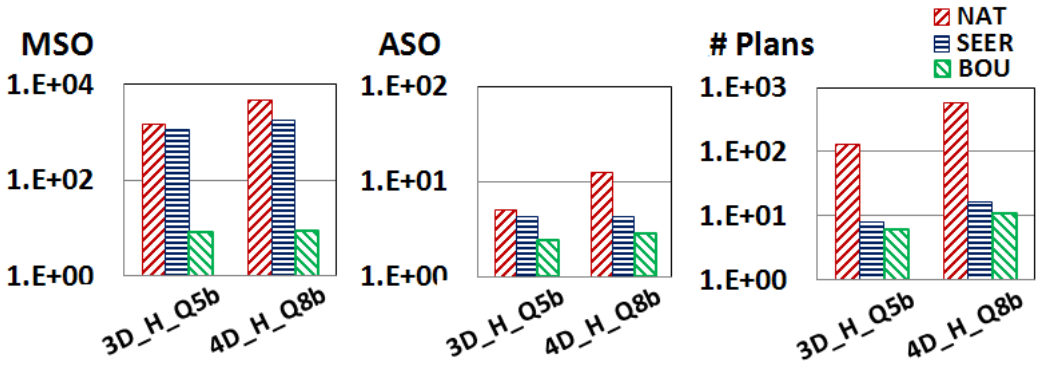

Figure 25: Commercial Engine Performance (log-scale)

7 Related Work

A rich body of literature is available pertaining to selectivity estimation issues [11]. We start with the overview of the closely related techniques which can be collectively termed as *plan-switching approaches*, as they involve run-time switching among complete query plans. At first glance, our bouquet approach, with its partial execution of multiple plans, may appear very similar to run-time re-optimization techniques such as POP [17] and Rio [4]. However, there are key differences: Firstly, they start with the optimizer's estimate as the initial seed, and then conduct a full-scale re-optimization if the estimate are found to be significantly in error. In contrast, we always start from the origin of the selectivity space, and directly choose plans from the bouquet for execution without invoking the optimizer again. A beneficial and unique side-effect of this start-from-origin approach is that it assures repeatability of the query execution strategy.

Secondly, both POP and Rio are based on heuristics and do not provide any performance bounds. In particular, POP may get stuck with a poor plan since its validity ranges are defined using structure-equivalent plans only. Similarly, Rio's sampling-based heuristics for monitoring selectivities may not work well for join-selectivities and its definition of plan robustness on the basis of performance at corners (principal diagonal) has not been justified.

Recently, a novel interleaved optimization and execution approach was proposed in [19] wherein plan fragments are selectively executed, when recommended by an error propagation framework, to guard

against the fallout of estimation errors. The error framework leverages an elegant histogram construction mechanism from [18] that minimizes the multiplicative error. While this technique substantively reduces the execution overheads, it provides no guarantees as it is largely based on heuristics.

Techniques that use a single plan during the entire query execution [9, 3, 13, 18, 6] run into the basic infeasibility of a single plan to be near-optimal across the entire selectivity space. The bouquet mechanism overcomes this problem by identifying a small set of plans that collectively provide the near-optimality property. Further, it does not require any prior knowledge of the query workload or the database contents. On the other hand, the use of only one active plan (at a time) to process the data makes the bouquet algorithm dissimilar from *Routing-based approaches* wherein different data segments may be routed to different simultaneously active plans – for example, plan per tuple [2] and plan per tuple group [20].

Our technique may superficially look similar to PQO techniques, (e.g. PPQO [5]), since a set of plans are identified before execution by exploring the selectivity space. The primary difference is that these techniques are useful for saving on optimization time for query instances with known parameters and selectivities. On the other hand, our goal is to regulate the worst case performance impact when the computed selectivities are likely to be erroneous.

Further, the bouquet technique does not modify plan structures at run-time (modulo spilling directives). This is a major difference from "plan-morphing" approaches, where the execution plan may be substantially modified at run-time using custom-designed operators, e.g. *chooseplan* [10], *switch* [4], *feedback* [7].

Finally, we emphasize that our goal of minimizing the worst case performance in the presence of unbounded selectivity errors, does not coincide with any of the earlier works in this area. Previously considered objectives include (a) improved performance compared to the optimizer generated plan [4, 13, 16, 17, 19]; (b) improved average performance and/or reduced variance [9, 6, 3]; (c) improved accuracy of selectivity estimation structures [1]; and (d) bounded impact of multiplicative estimation errors [18].

8 Critique of Bouquet Approach

Having presented the mechanics and performance of the bouquet approach, we now take a step back and critique the technique.

The bouquet approach is intended for use in difficult estimation environments – that is, in database setups where accurate selectivity estimation is hard to achieve. However, when estimation errors are apriori known to be small, re-optimization techniques such as [17, 4], which use the optimizer's estimate as the initial seed, are likely to converge much quicker than the bouquet algorithm, which requires starting at the origin to ensure the first quadrant invariant. But, if the estimates were apriori guaranteed to be *under-estimates*, then the bouquet algorithm can also leverage the initial seed.

Being a *plan-switching* approach, the bouquet technique suffers from the drawbacks generic to such approaches: Firstly, they are poor at serving *latency-sensitive* applications as they have to perforce wait for the final plan execution to return result tuples. Secondly, they are not recommended for update queries since maintaining transactional consistency with multiple executions may incur significant overheads to rollback the effects of the aborted partial executions. Finally, with single-plan optimizers, DBAs use their domain knowledge to fine-tune the plan using "plan-hints". But this is not straightforward in *plan-switching* techniques since the actual plan sequence is determined only at run-time. Notwithstanding the limitations, such techniques are now featured even in commercial products (e.g. [27]).

There are also a few problems that are *specific* to the bouquet approach: Firstly, while it is inherently robust to changes in data *distribution*, since these changes only shift the location of q_a in the existing ESS, the same is not true with regard to database *scale-up*. That is, if the database size increases significantly, then the original ESS no longer covers the entire error space. An obvious solution to handle this problem is to recompute the bouquet from scratch, but most of the processing may turn out to be redundant. Therefore, developing incremental bouquet maintenance strategies is an interesting future research challenge.

Secondly, the bouquet identification overheads increase exponentially with dimensionality. Apart from the obvious amortization over repeated query invocations, we also described some mechanisms for reducing these overheads in Section 6.3. Further, a complex query does not necessarily imply a commensurately large number of error dimensions because: (i) The selectivities of base relation predicates of the form "*column op constant*" can be estimated accurately with current techniques; (ii) The join-selectivities for PK-FK joins can be estimated accurately if the entire PK-relation participates in the join; (iii) The partial derivatives of the POSP plan cost functions along each dimension can be computed on a low resolution mapping of the ESS, and any dimension with a small derivative across all the plans can be eliminated since its cost impact is marginal.

Thirdly, the identification of ESS dimensions may not always be straightforward. For example, in cyclic queries, different plans may combine predicates in different ways. One option to handle this scenario is to first construct the ESS using individual predicates as dimensions. Then, assuming that predicate independence holds, the selectivity of any predicate combination could be *inferred* using the existing values for the individual constituent predicates.

Given the above discussion, the bouquet approach is currently recommended specifically for providing response-time robustness in large archival read-only databases supporting complex decision-support applications that are likely to suffer significant estimation errors. We expect that many of today's OLAP installations may fall into this category.

9 Conclusions

Selectivity estimation errors resulting in poor query processing performance is part of the database folklore. In this paper, we investigated a new approach to this classical problem, wherein the estimation process was completely jettisoned for error-prone predicates. Instead, such selectivities were explicitly and progressively discovered at run-time through a carefully graded sequence of partial executions from a "plan bouquet". The execution sequence, which followed a cost-doubling geometric progression, ensured that the overheads are bounded, thereby limiting the MSO incurred by the execution to 4 times the plan cardinality of the densest isocost contour. To the best of our knowledge, such bounds have not been previously presented in the database literature.

To ensure that the actual overheads in practice were much lower than the worst-case bound values, we also proposed the Anorexic Reduction, AxisPlans and Spilling optimizations for minimizing the bouquet size, minimizing the number of partial executions, and maximizing the selectivity movement in each execution, respectively. Their collective benefits ensured that MSO was less than 10 across all the queries in our evaluation set, an enormous improvement compared to the performance of the native optimizer, wherein this metric ranged from the thousands to the millions. Further, the optimizations also ensured that the bouquet's ASO performance was always either comparable to or much better than the native optimizer. Finally, while the bouquet algorithm did occasionally perform worse than the native optimizer for specific query locations, such situations occurred at less than 1% of the locations, and the performance degradation was relatively small, a factor of 3 or less.

Overall, the bouquet approach promises guaranteed performance and repeatability in query execution, features that had hitherto not been available, thereby opening up new possibilities for robust query processing.

Acknowledgments. We thank the anonymous reviewers and S. Sudarshan, Prasad Deshpande, Srinivas Karthik, Sumit Neelam and Bruhathi Sundarmurthy for their constructive comments on this work.

References

[1] A. Aboulnaga and S. Chaudhuri, "Self-tuning Histograms: Building Histograms without Looking at Data", *ACM SIGMOD Conf.*, 1999.

[2] R. Avnur and J. Hellerstein, "Eddies: Continuously Adaptive Query Processing", *ACM SIGMOD Conf.*, 2000.

[3] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach", *ACM SIGMOD Conf.*, 2005.

[4] S. Babu, P. Bizarro and D. DeWitt, "Proactive Re-Optimization", *ACM SIGMOD Conf.*, 2005.

[5] P. Bizarro, N. Bruno, D. Dewitt, "Progressive Parametric Query Optimization", *IEEE TKDE*, 21(4), 2009.

[6] S. Chaudhuri, H. Lee and V. Narasayya, "Variance aware optimization of parameterized queries", *ACM SIGMOD Conf.*, 2010.

[7] S. Chaudhuri, V. Narasayya and R. Ramamurthy, "A Pay-As-You-Go Framework for Query Execution Feedback", *PVLDB*, 1(1), 2008.

[8] M. Chrobak, C. Kenyon, J. Noga and N. Young, "Incremental Medians via Online Bidding", *Algorithmica*, 50(4), 2008.

[9] F. Chu, J. Halpern and J. Gehrke, "Least Expected Cost Query Optimization: What Can We Expect", *ACM PODS Conf.*, 2002.

[10] R. Cole and G. Graefe, "Optimization of Dynamic Query Evaluation Plans", *ACM SIGMOD Conf.*, 1994.

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

[12] G. Graefe et al, "Robust Query Processing (Dagstuhl Seminar 12321)", *Dagstuhl Reports*, 2(8), 2012.

[13] Harish D., P. Darera and J. Haritsa, "Identifying Robust Plans through Plan Diagram Reduction", *PVLDB*, 1(1), 2008.

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

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

[16] N. Kabra and D. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans", *ACM SIGMOD Conf.* 1998.

[17] V. Markl et al, "Robust Query Processing through Progressive Optimization", *ACM SIGMOD Conf.*, 2004.

[18] G. Moerkotte, T. Neumann and G. Steidl, "Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors", *PVLDB*, 2(1), 2009.

[19] T. Neumann and C. Galindo-Legaria, "Taking the Edge off Cardinality Estimation Errors using Incremental Execution", *BTW Conf.*, 2013.

[20] N. Polyzotis, "Selectivity-based partitioning: A Divide and Union Paradigm for Effective Query Optimization", *ACM CIKM Conf.*, 2005.

[21] P. Selinger et al, "Access Path Selection in a Relational Database Management System", *ACM SIGMOD Conf.*, 1979.

[22] M. Stillger, G. Lohman, V. Markl and M. Kandil, "LEO – DB2's LEarning Optimizer", *VLDB Conf.*, 2001.

[23] W. Wu et al, "Predicting Query Execution Times: Are Optimizer Cost Models Really Usable?", *IEEE ICDE Conf.*, 2013.

[24] G. Lohman, "Is Query Optimization a "solved" problem?", *ACM SIGMOD Blog (April 2014)* `http://wp.sigmod.org/?author=20`

[25] `technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx`

[26] `www.ibm.com/developerworks/data/library/tips/dm-0312yip/`

[27] `www.oracle.com/ocom/groups/public/@otn/documents/webcontent/1963236.pdf`

[28] `www.postgresql.org/docs/8.4/static/release.html`

[29] `doxygen.postgresql.org/structInstrumentation.html`

10 Appendix

10.1 Query Text (based on benchmark queries)

```
select
        n_name,
        l_extendedprice * (1 - l_discount) as revenue
from
        customer, orders, lineitem, supplier, nation, region
where
        c_custkey = o_custkey and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey and n_regionkey = r_regionkey
        and o_orderdate >= 1994-01-01
        and o_orderdate < 1994-01-01 + interval '25' day
        and c_acctbal <= 9900 and s_acctbal <= 9900
```

Figure 26: **3D_H_Q5 (Based on TPC-H Query 5)**

```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer, orders, lineitem, supplier, nation, region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'
        and o_totalprice ≤ $X1
        and c_acctbal ≤ $X2
        and l_extendedprice ≤ $X3
group by
        n_name
order by
        revenue desc
```

Figure 27: **3D_H_Q5^b (Based on TPC-H Query 5)**

.

```
select
        supp_nation, cust_nation, l_year, volume
from (
        select
                n1.n_name as supp_nation, n2.n_name as cust_nation,
                extract(year from l_shipdate) as l_year, l_extendedprice * (1- l_discount)
                as volume
        from
                supplier,lineitem, orders, customer, nation n1, nation n2
        where
                s_suppkey = l_suppkey and o_orderkey = l_orderkey
                and c_custkey = o_custkey and s_nationkey = n1.n_nationkey
                and c_nationkey = n2.n_nationkey
                and l_shipdate between date '1995-01-01' and date '1996-12-31'
                and c_acctbal <= 9900 and s_acctbal <= 9900 )
```

Figure 28: **3D_H_Q7 (Based on TPC-H Query 7)**

```
select
        supp_nation, cust_nation, l_year, volume
from (
        select
                n1.n_name as supp_nation, n2.n_name as cust_nation,
                extract(year from l_shipdate) as l_year, l_extendedprice * (1- l_discount)
                as volume
        from
                supplier,lineitem, orders, customer, nation n1, nation n2
        where
                s_suppkey = l_suppkey and o_orderkey = l_orderkey
                and c_custkey = o_custkey and s_nationkey = n1.n_nationkey
                and c_nationkey = n2.n_nationkey
                and ( (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
                or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE') )
                and l_shipdate between date '1995-01-01' and date '1996-12-31'
                and c_acctbal <= 9900 and s_acctbal <= 9900 )
```

Figure 29: **5D_H_Q7 (Based on TPC-H Query 7)**

.

```
select
        o_year, volume
from (
        select
                extract(year from 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 o_orderdate between date '1995-01-01' and date '1995-09-01'
                and p_type = 'ECONOMY ANODIZED STEEL'
                and c_acctbal <= 9900 and s_acctbal <= 9900 )
```

Figure 30: **4D_H_Q8 (Based on TPC-H Query 8)**

.

```
select
    o_year,
    sum(case when nation = 'BRAZIL' then volume
    else 0 end) / sum(volume) as mkt_share
from
    (
    select
        DATE_PART('YEAR',o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) as volume,
        n2.n_name as nation
    from
        part, supplier, lineitem, orders,
        customer, nation n1, nation n2, region
    where
        p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = 'AMERICA'
        and s_nationkey = n2.n_nationkey
        and p_retailprice ≤ $X1
        and s_acctbal ≤ $X2
        and l_extendedprice ≤ $X3
        and o_totalprice ≤ $X4
    ) as all_nations
group by
    o_year
order by
    o_year
```

Figure 31: **4D_H_Q8^b (Based on TPC-H Query 8)**

.

```
select
        i_item_id, ss_quantity, ss_list_price,
        ss_coupon_amt, ss_sales_price
from
        store_sales, customer_demographics, date_dim, item, promotion
where
        ss_sold_date_sk = d_date_sk and ss_item_sk = i_item_sk and
        ss_cdemo_sk = cd_demo_sk and ss_promo_sk = p_promo_sk and
        cd_gender = 'F' and cd_marital_status = 'M' and cd_education_status = 'College'
        and (p_channel_email = 'N' or p_channel_event = 'N') and d_year = 2001
        and i_current_price < 99 and p_cost <= 1000
```

Figure 32: **4D_DS_Q7 (Based on TPC-DS Query 7)**

```
select
        ca_zip, cs_sales_price
from
        catalog_sales, customer, customer_address, date_dim
where
        cs_bill_customer_sk = c_customer_sk and c_current_addr_sk = ca_address_sk
        and ( substr(ca_zip,1,5) in ('85669', '86197','88274', '83405',
        '86475', '85392', '85460', '80348', '81792')
        or ca_state in ('CA','WA','GA'))
        and cs_sold_date_sk = d_date_sk and d_qoy = 2 and d_year = 1999
```

Figure 33: **3D_DS_Q15 (Based on TPC-DS Query 15)**

```
select
        i_brand_id brand_id, i_brand brand, i_manufact_id,
        i_manufact, ss_ext_sales_price
from
        date_dim, store_sales, item, customer, customer_address, store
where
        d_date_sk = ss_sold_date_sk and ss_item_sk = i_item_sk
        and i_manager_id=97 and d_moy=12 and d_year=2002
        and ss_customer_sk = c_customer_sk and c_current_addr_sk
        =ca_address_sk and substr(ca_zip,1,5) <> substr(s_zip,1,5)
        and ss_store_sk = s_store_sk
        and s_tax_percentage <= 0.1
```

Figure 34: **5D_DS_Q19 (Based on TPC-DS Query 19)**

.

```
select
        i_item_id, avg(cs_quantity) , avg(cs_list_price) ,
        avg(cs_coupon_amt) , avg(cs_sales_price)
from
        catalog_sales, customer_demographics, date_dim, item, promotion
where
        cs_sold_date_sk = d_date_sk and cs_item_sk = i_item_sk and
        cs_bill_cdemo_sk = cd_demo_sk and cs_promo_sk = p_promo_sk
        and cd_gender = 'F' and cd_marital_status = 'U' and cd_education_status
        = Unknown' and (p_channel_email = 'N' or p_channel_event = 'N') and
        d_year = 2002 and i_current_price <= 99
group by
        i_item_id
order by
        i_item_id
```

Figure 35: **4D_DS_Q26 (Based on TPC-DS Query 26)**

```
select
        cc_call_center_id , cc_name , cc_manager , sum(cr_net_loss)
from
        call_center,catalog_returns, date_dim, customer, customer_address,
        customer_demographics, household_demographics
where
        cr_call_center_sk = cc_call_center_sk and cr_returned_date_sk =
        d_date_sk and cr_returning_customer_sk= c_customer_sk and cd_demo_sk
        =c_current_cdemo_sk and hd_demo_sk = c_current_hdemo_sk and
        ca_address_sk = c_current_addr_sk and d_year = 2000 and d_moy = 12
        and ( (cd_marital_status = 'M' and cd_education_status = 'Unknown')
        or(cd_marital_status = 'W' and cd_education_status = 'Advanced Degree'))
        and hd_buy_potential like '5001-10000%' and ca_gmt_offset = -7

group by
        cc_call_center_id,cc_name,cc_manager,cd_marital_status, cd_education_status

order by
        sum(cr_net_loss) desc
```

Figure 36: **4D_DS_Q91 (Based on TPC-DS Query 91)**

.

```
select   s_store_name, hd_dep_count, ss_list_price, s_company_name
from
        store_sales, household_demographics, time_dim, store
where
        ss_sold_time_sk = time_dim.t_time_sk and
        ss_hdemo_sk = household_demographics.hd_demo_sk and
        ss_store_sk = s_store_sk and time_dim.t_hour = 8
        and time_dim.t_minute >= 30 and
        household_demographics.hd_dep_count = 2
        and store.s_store_name = 'ese'
```

Figure 37: **3D_DS_Q96 (Based on TPC-DS Query 96)**

.

10.2 Query Join Graphs (Experiments)

Here, we show the query join graphs with base-relation selectivities, the error-prone selectivities (shown in dark red) and error-free selectivities (shown in dark green). There are ten queries involving chain, branch and star graphs from TPCH and TPCDS schemas.

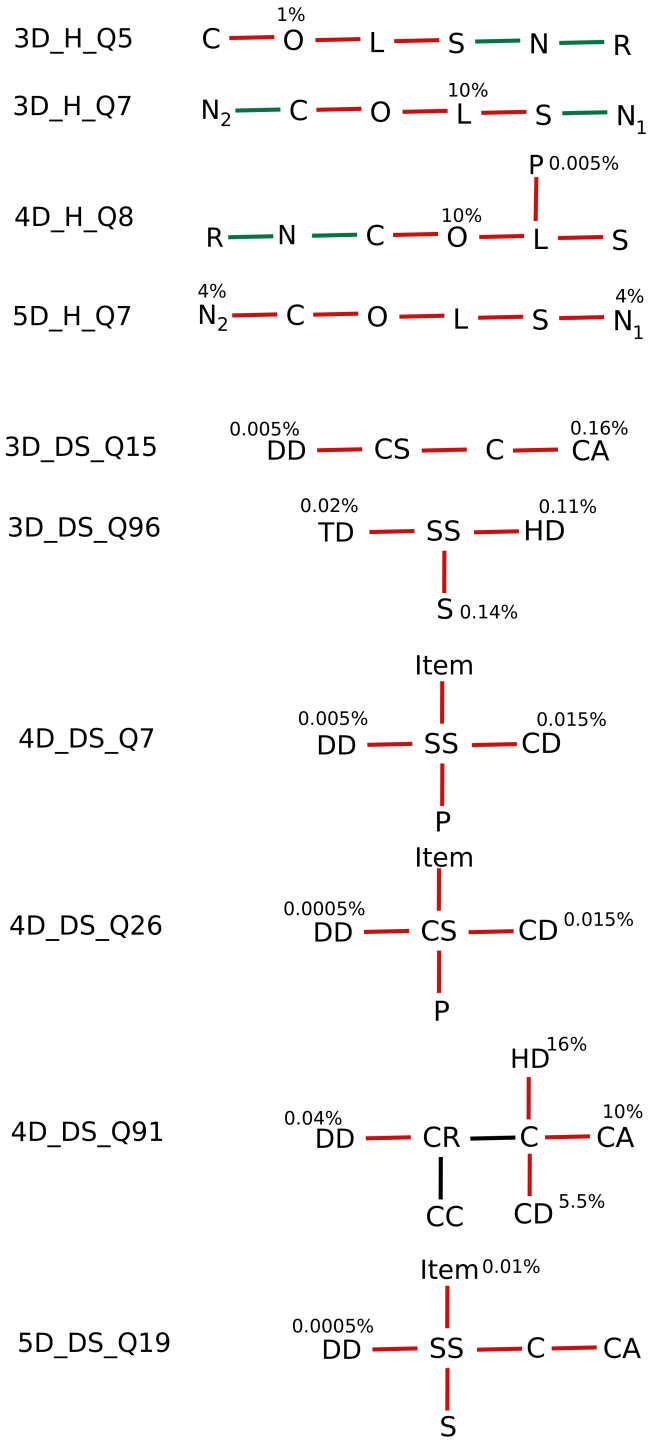

Figure 38: Query Join Graphs