Plan Bouquets: A Fragrant Approach to Robust Query Processing

ANSHUMAN DUTT and JAYANT R. HARITSA, Indian Institute of Science

Identifying efficient execution plans for declarative OLAP queries typically entails estimation of several predicate selectivities. In practice, these estimates often differ significantly from the values actually encountered during query execution, leading to poor plan choices and grossly inflated response times. We propose here a conceptually new approach to address this classical problem, wherein the compile-time estimation process is completely eschewed for error-prone selectivities. Instead, from the set of optimal plans in the query's selectivity error space, a limited subset, called the "plan bouquet," is selected such that at least one of the bouquet plans is 2-optimal at each location in the space. Then, at run time, a sequence of cost-budgeted executions from the plan bouquet is carried out, eventually finding a plan that executes to completion within its assigned budget. The duration and switching of these executions is controlled by a graded progression of isosurfaces projected onto the optimal performance profile. We prove that this construction results, for the first time, in guarantees on worst-case performance sub-optimality. Moreover, it ensures repeatable execution strategies across different invocations of a query.

We then present a suite of enhancements to the basic plan bouquet algorithm, including randomized variants, that result in significantly stronger performance guarantees. An efficient isosurface identification algorithm is also introduced to curtail the bouquet construction overheads.

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 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, it can be implemented using existing optimizer infrastructure, making it relatively easy to incorporate in current database engines.

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

CCS Concepts: • **Information systems → Query optimization;**

Additional Key Words and Phrases: Selectivity estimation, plan bouquets, robust query processing

ACM Reference Format:
Anshuman Dutt and Jayant R. Haritsa. 2016. Plan bouquets: A fragrant approach to robust query processing. ACM Trans. Database Syst. 41, 2, Article 11 (May 2016), 37 pages.
DOI: http://dx.doi.org/10.1145/2901738

11

1. INTRODUCTION

Cost-based database query optimizers estimate a host of *selectivities* while constructing efficient execution plans for declarative online analytical processing (OLAP) queries. For example, consider **EQ**, the simple select-project-join (SPJ) query shown in Figure 1 for enumerating orders of cheap parts—here, the optimizer estimates the selectivities of a selection predicate ($p\_retailprice < 1000$) and two join predicates ($part \bowtie lineitem$, $orders \bowtie lineitem$). In practice, these estimates are often significantly in error with

A preliminary version of this article was published in the ACM SIGMOD 2014 conference.
Authors' addresses: A. Dutt and J. R. Haritsa, Database Systems Lab, SERC/CSA, Indian Institute of Science, Bangalore 560012, India; emails: {anshuman, haritsa}@dsl.serc.iisc.ernet.in.

DOI: http://dx.doi.org/10.1145/2901738

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

Fig. 1. Example Query (EQ).

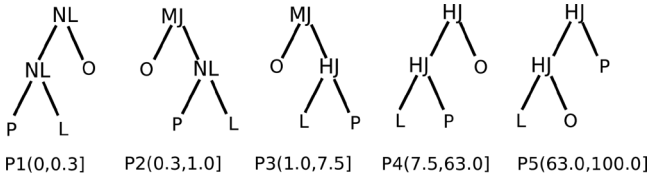

Fig. 2. POSP plans on p_retailprice dimension.

respect to the actual values subsequently encountered during query execution. Such errors, which can even be in *orders of magnitude* in real database environments [Markl et al. 2004; Lohman 2014], arise due to a variety of well-documented reasons [Stillger et al. 2001; Lohman 2014], including outdated statistics, coarse summaries, attribute-value independence (AVI) assumptions, complex user-defined predicates, and error propagation in the query execution operator tree [Ioannidis and Christodoulakis 1991]. 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 [Selinger et al. 1979]). The net outcome of these erroneous estimates is that the execution plans recommended by the query optimizer may turn out to be poor choices at runtime, resulting in grossly inflated query response times.

A considerable body of literature exists on proposals to tackle this classical problem. For instance, techniques for improving the *statistical quality* of the metadata include improved summary structures [Aboulnaga and Chaudhuri 1999; Moerkotte et al. 2009], feedback-based adjustments [Stillger et al. 2001], and on-the-fly re-optimization of queries [Kabra and DeWitt 1998; Babu et al. 2005; Neumann and Galindo-Legaria 2013]. A complementary approach is to identify *robust plans* that are relatively less sensitive to estimation errors [Chu et al. 2002; Babcock and Chaudhuri 2005; Babu et al. 2005; Harish et al. 2008]. While these prior techniques provide novel and innovative formulations, a common limitation is their inability to furnish *performance guarantees*.

Plan Bouquet Approach

In this article, 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 runtime through a calibrated sequence of cost-limited plan executions. That is, we attempt to sidestep the selectivity estimation problem, rather than address it head-on, by adopting a *"seeing is believing"* perspective on these values.

One-Dimensional Example. We introduce the new approach through a restricted one-dimensional (1D) version of the EQ example query wherein only the $p\_retailprice < 1000$ selection predicate is error-prone. First, through repeated invocations of the optimizer, we identify the "parametric optimal set of plans" (POSP) that covers 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

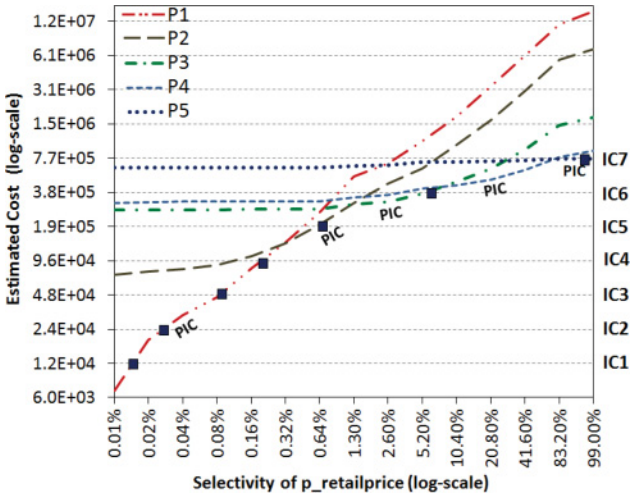

Fig. 3. POSP performance (log-log scale).

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.)

The optimizer-computed 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 exercise is carried out at query compilation time. Subsequently, at runtime, the correct query selectivities are *implicitly* discovered through a sequence of *cost-limited* executions of bouquet plans. Specifically, beginning with the cheapest cost step, we iteratively execute the bouquet plan assigned to each step until either:

(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-optimal* 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 P1 are *discarded*. The new plan P2 is executed until the associated overhead limit (1.9E5) is reached. The cost horizon is now extended to IC6 (3.8E5 | 6.5%), in the process discarding P2's intermediate results and executing P3 instead. The execution in this case 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) employed for the final execution.

Extension to Multiple Dimensions. When the above 1D approach is generalized to a 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 2D case, the IC steps are horizontal planes cutting through a hollow three-dimensional PIC surface, typically resulting in hyperbolic intersection contours featuring a multitude of plans covering disjoint segments of the contours—an instance of this scenario is shown in Figure 6.

Notwithstanding these changes, the basic mechanics of the bouquet algorithm remain virtually identical. The primary difference is that we jump from one isosurface to the next only after it is determined that *none* of the bouquet plans present on the current isosurface 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 runtime, 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 article, 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, we define robustness to be "the worst-case sub-optimality in plan performance that can arise due to selectivity errors," denoted as MSO (maximum sub-optimality). With respect to this MSO metric, the bouquet mechanism delivers substantial improvements over current optimizers. Moreover, it does so while providing comparable or improved average-case performance.

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), which is comprised of the supremum of the individual plan profiles. In fact, the MSO 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 small 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 article are incorporated, the enhanced bouquet's performance (dashed blue) improves to 3.1 (worst case) and 1.7 (average case), thereby dominating the native optimizer on both metrics.

Performance Guarantees and Enhancements

Our motivation for the cost-based discretization of the PIC is that it leads to *guaranteed* bounds on MSO. For instance, we prove that the cost-doubling strategy used in the 1D example results in an *MSO upper-bound of 4*—this bound is inclusive of

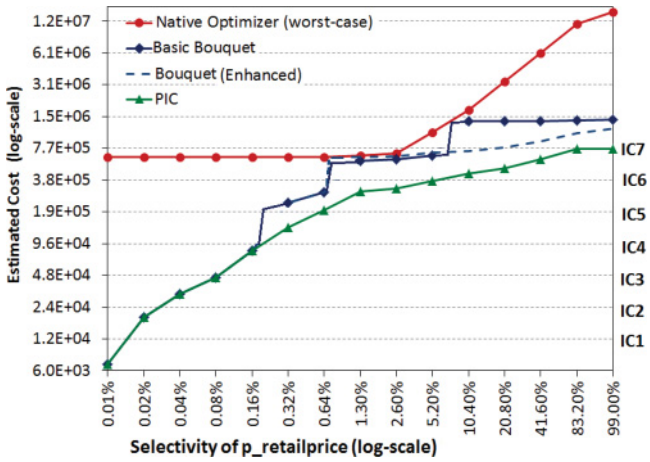

Fig. 4. Bouquet performance (log-log scale).

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 MSO bound becomes 4 times the bouquet cardinality (more accurately, the plan cardinality of the densest isosurface). 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 [Chrobak et al. 2008]).

We also propose a suite of enhancements to the basic plan bouquet algorithm that result in significantly stronger performance guarantees. Specifically, compile-time enhancements that help to reduce the *effective* isosurface plan densities, and randomization strategies that improve on the maximum expected sub-optimality. Moreover, an efficient isosurface identification algorithm is introduced to curtail the bouquet construction overheads. Finally, with regard to implementation, these techniques can be largely constructed using only API features (e.g., abstract plan costing) that have already found expression in modern DB engines, as explained later in Section 7.

Experimental Evaluation

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 MSO plummeted from about 10^6 to just 10! The potency of the approach is also indicated by its providing an MSO guarantee of less than 20 over our entire query workload, while the average sub-optimality was typically within a factor of 4 wrt to the optimal.

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, the bouquet mechanism has another major benefit: On a given database, the execution strategy for a particular query instance, that is, 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 [Babcock and Chaudhuri 2005].

In our presentation 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. This issue was recently investigated in Wu et al. [2013], where a cost model tuning mechanism was proposed that enabled prediction of execution times within an error threshold δ of around 40%. Given such a bounded δ, it is proved in Dutt and Haritsa [2014b] that the MSO guarantees of the plan bouquet approach continue to hold after inflation by a $(1+\delta)^2$ factor.

In closing, we wish to highlight that, from a deployment perspective, the bouquet technique is intended to *complementarily co-exist* with the classical optimizer setup and not to replace it. It is left 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 Section 10.

Organization. The remainder of the article is organized as follows: In Section 2, a precise description of the robust execution problem is provided, along with associated notations. Theoretical bounds on the MSO provided by the bouquet technique are presented in Section 3. Corresponding bounds on the maximum expected sub-optimality with randomized variants of the algorithm are derived in Section 4. A variety of compile-time enhancements that result in significantly stronger MSO guarantees are described in Section 5. An efficient mechanism to achieve pragmatic overheads for bouquet identification is outlined in Section 6, followed by other implementation details of the plan bouquet architecture in Section 7. The experimental framework and performance results are reported in Section 8. Related work is reviewed in Section 9, while Section 10 presents a critical review of the bouquet approach. Finally, we conclude in Section 11.

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 [Graefe et al. 2012]—here, we use the notion of *performance sub-optimality* to characterize robustness.

Error-Prone Selectivity Space

In our framework, each user query Q is associated with a set of selectivity predicates SP, a subset of which are error-prone wrt their estimation. Next, we define a query space QS for Q to be $\{Q, AKP, EPP\}$, where AKP is the set of predicates with accurately known selectivities, and EPP is comprised of the remaining error-prone predicates (i.e., $AKP \cup EPP = SP$). From the EPP, we construct an *error-prone selectivity space*, called ESS, wherein each error-prone predicate maps to an independent $[0, 1]$ selectivity dimension in the space. That is, ESS is a $[0, 1]^D$ hypercube with $D = |EPP|$, where each D-dimensional point $q(s_1, s_2, \ldots, s_D)$ represents a possible location of the query Q, as determined by its selectivities on each of these dimensions. The assignment of an independent dimension to each EPP is in conformity with the *selectivity independence* assumption that is prevalent in modern query optimizer frameworks.

In the ESS defined as above, the cost of an execution plan P_i at a query location q in the ESS is denoted by $c(P_i, q)$. Also, we denote the query optimizer's *estimated* location of Q in the ESS by q_e and the *actual* location at runtime by q_a. The optimal plan at q_e, as determined by the native optimizer, is denoted by $P_{opt}(q_e)$, and, similarly, the optimal plan at q_a by $P_{opt}(q_a)$. Further, we assume that the query locations and the associated estimation errors range over the *entire* ESS, that is, all (q_e, q_a) error combinations are possible.

Sub-Optimality-Based Robustness Metrics

With the above query model, the sub-optimality incurred due to using plan $P_{opt}(q_e)$ at location q_a is simply defined as the ratio:

$$SubOpt(q_e, q_a) = \frac{c(P_{opt}(q_e), q_a)}{c(P_{opt}(q_a), q_a)} \quad \forall q_e, q_a \in ESS \tag{1}$$

with $SubOpt$ ranging over $[1, \infty)$. The worst-case $SubOpt$ for a given 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) error combination that results in the maximum value of $SubOpt$ over the entire ESS, that is:

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

The above definitions are appropriate for the manner in which modern optimizers operate, wherein selectivity estimates are made at compile time, and a single plan is executed at runtime. 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 runtime. Notwithstanding, we can still compute the corresponding statistics by: (a) substituting q_e with a "don't care" $*$ and (b) having the cost of the bouquet, denoted by $c(B, q_a)$, include the overheads incurred by the exploratory partial executions. That is,

$$SubOpt(*, q_a) = \frac{c(B, q_a)}{c(P_{opt}(q_a), q_a)} \quad \forall q_a \in ESS \tag{4}$$

and

$$\mathbf{MSO} = \max_{q_a \in ESS} (SubOpt(*, q_a)). \tag{5}$$

Finally, the bouquet technique also furnishes a *guarantee* on its MSO performance, which is denoted by $\mathbf{MSO_g}$.

Analogously to the above, the *randomized* variants of the bouquet algorithm are evaluated for the *maximum expected sub-optimality* across the ESS, defined as

$$\mathbf{MESO} = \max_{q_a \in ESS} (E[SubOpt(*, q_a)]),$$

and the guarantee on maximum expected sub-optimality is denoted by $\mathbf{MESO_g}$.

Ancillary Performance Metrics

In addition to the above primary metrics, we also evaluate the bouquet technique over a related set of performance metrics. Specifically, if we assume that all query locations and error combinations are equally likely, that is, the estimated query locations and

Table I. Reference Table for Notations

Notation	Description
Q	User query
ESS	Error-prone Selectivity Space
D	Number of ESS dimensions
$q(s_1, s_2, \ldots, s_D)$	Query Location in ESS
q_e	Optimizer estimated selectivity location in ESS
q_a	ESS location corresponding to actual runtime selectivities
$P_{opt}(q)$	Optimal plan at location q
$c_{opt}(q)$	Cost of optimal plan at location q
$c_B(q)$	Cost incurred by plan bouquet for location q
$c(P_i, q)$	Cost of plan P_i at location q
$SubOpt_{worst}(q_a)$	Worst-case native sub-optimality for location q_a
$SubOpt(*, q_a)$	Sub-optimality of location q_a for plan bouquet q_a
MSO	Worst-case sub-optimality across ESS
ASO	Average sub-optimality across ESS
MH	Maximum harm across ESS
MSO_g	Compile-time guarantee on worst-case sub-optimality
$MESO_g$	Compile-time guarantee on maximum expected sub-optimality
IC_k	k^{th} isosurface (isocost surface) in the ESS
cost(IC_k)	Cost-budget corresponding to isosurface IC_k

the actual query locations are uniformly and independently distributed over the entire ESS, the *average* sub-optimality over ESS is defined as:

$$\textbf{ASO} = \frac{\sum_{q_e \in ESS} \sum_{q_a \in ESS} SubOpt(q_e, q_a)}{\sum_{q_e \in ESS} \sum_{q_a \in ESS} 1}. \tag{6}$$

The corresponding version for the bouquet technique is

$$\textbf{ASO} = \frac{\sum_{q_a \in ESS} SubOpt(*, q_a)}{\sum_{q_a \in ESS} 1}. \tag{7}$$

These definitions can easily be extended to the general case where the estimated and actual locations have idiosyncratic probability distributions.

An important point to note is that even when the bouquet algorithm performs well on the MSO and ASO metrics, it is possible that for some specific locations $q_a \in$ ESS, its performance is poorer than the worst performance of the native optimizer—that is, the bouquet is *harmful* for the queries associated with these locations. This possibility is captured using the following *MaxHarm* metric:

$$\textbf{MH} = \max_{q_a \in ESS} \left(\frac{SubOpt(*, q_a)}{SubOpt_{worst}(q_a)} - 1 \right). \tag{8}$$

Note that MH values lie in the range $(-1, \text{MSO}_g - 1]$ and harm occurs whenever MH is positive.

For notational convenience, we will hereafter represent the optimal cost and the bouquet cost for a given location q with $c_{opt}(q)$ and $c_B(q)$, respectively. Inclusive of these, the common notations used in the article are enumerated in Table I for quick reference.

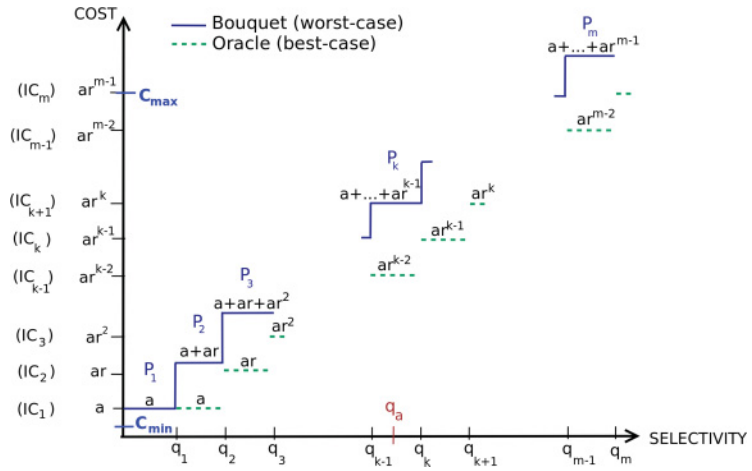

Fig. 5. 1D selectivity space.

Plan Cost Functions

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. It captures the intuitive observation that when more data are processed by a plan, signified by larger selectivities, the cost of processing also increases. This assumption has often been made in the literature [Bizarro et al. 2009; Chaudhuri et al. 2010; Harish et al. 2007] and generally holds for the plans generated by current database systems on decision-support queries [Reddy and Haritsa 2005]. The only exception that 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 having optimal cost surfaces with a maxima or minima in the *interior* of the error space are not amenable to our approach.

Apart from monotonicity, we also assume the cost functions to be *continuous (smooth)* throughout the ESS, again, a commonplace feature in practice.

3. ROBUSTNESS BOUNDS

We begin our presentation of the plan bouquet approach by characterizing its MSO performance bounds for the 1D scenario. Subsequently, we extend the analysis to the general multi-dimensional case.

3.1. 1D Selectivity Space

By virtue of our assumptions on plan cost behavior, the PIC is a monotonically increasing and continuous function throughout the ESS; its minimum and maximum costs are denoted by C_{min} and C_{max}, respectively. As described in the Introduction, this PIC is discretized by projecting a graded progression of cost steps onto the curve. Specifically, consider the case wherein the 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 $m = \lfloor log_r \frac{C_{max}}{C_{min}} + 1 \rfloor$ cuts, $IC_1, IC_2, \ldots, IC_m$, satisfying the boundary conditions $a/r < C_{min} \leq cost(IC_1) = a$ and $cost(IC_{m-1}) < C_{max} = cost(IC_m)$, as shown in Figure 5.

For $1 \leq k \leq m$, denote the selectivity location where the k^{th} cost step (IC_k) intersects the PIC by q_k and the corresponding bouquet plan as P_k. All the q_k locations are unique,

ALGORITHM 1: 1D Bouquet Algorithm

```
// for each cost step IC_k
for k = 1 to m do
    start executing bouquet plan P_k
    // perform cost-limited execution
    while run_cost(P_k) ≤ cost(IC_k) do
        execute P_k
        if P_k completes execution then
            return query result
        end
    end
    terminate P_k and discard partial results
end
```

by definition, due to the monotonicity and continuity features of the PIC. 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.

With this framework, the bouquet execution algorithm, outlined in Algorithm 1, 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 $cost(IC_1)$, 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 $cost(IC_k)$. It is easy to see that the following lemma holds:

LEMMA 3.1. *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. 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:

$$c_B(q_a) = cost(IC_1) + cost(IC_2) + \cdots + cost(IC_k)$$

$$c_B(q_a) = a + ar + ar^2 + \cdots + ar^{k-1} = \frac{a(r^k - 1)}{r - 1}. \tag{9}$$

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

$$SubOpt(*, q_a) \leq \frac{\frac{a(r^k-1)}{r-1}}{ar^{k-2}} = \frac{r^2}{r-1} - \frac{r^{2-k}}{r-1} < \frac{r^2}{r-1}. \tag{10}$$

Note that the final 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 3.2. *Given a query* Q *with a 1D* ESS*, and the associated PIC discretized with a geometric progression having common ratio r, the bouquet execution algorithm ensures that* $MSO_g = \frac{r^2}{r-1}$.

Further, the choice of r can be optimized to minimize this value—the right-hand side reaches its minima at $r = 2$, at which the value of MSO_g 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 3.3. *Given a universe of cost-limited executions of POSP plans, no deterministic online algorithm can ensure* MSO_g *lower than 4 in the 1D scenario.*

PROOF. We prove by contradiction, assuming there exists an optimal online robust algorithm, R^* with a MSO_g of f, $f < 4$.

The proof is divided into two parts: First, we show that R^* must be a monotonically increasing sequence of plan execution costs, $[a_1, a_2, \ldots, a_m]$, and, second, we demonstrate that achieving an MSO of less than 4 requires the ratio of cumulative costs for consecutive steps in the sequence to be strictly decreasing; however, this is fundamentally impossible and hence the contradiction.

(a) (a) Assume that R^* has cost sequence $[a_1, \ldots, a_i, a_j, \ldots, a_{m+1}]$ that is sorted in increasing order except for the inversion caused by $a_j < a_i$.

Now, let us define a plan execution to be *useful* if its execution covers a hitherto uncovered region of the selectivity space. With this definition, an execution of a_j after a_i is clearly useless since no fresh selectivity ground is covered by this cheaper execution. A sample instance with reference to Figure 5 is executing P_2, which covers the selectivity region $(0, q_2)$, after P_3, which covers the region $(0, q_3)$; this does not add any value since the latter subsumes the former.

In summary, an out-of-order execution sequence cannot provide *any* improvement over an ordered sequence, which is why a_j can be safely discarded to give a completely sorted sequence $[a_1, \ldots, a_i, \ldots, a_m]$.

(b) For the sorted execution sequence R^*, denote the cumulative cost at each step with $A_j = \sum_{i=1}^{j} a_i$ and the *ratio* between the cumulative costs for consecutive steps as $Y_j = \frac{A_{j+1}}{A_j}$. Note that, by definition, $A_{j+1} > A_j$.

Now, since R^* has MSO_g of f, the sub-optimality caused by each and every step should be at most f, that is,

$$\frac{A_{j+1}}{a_j} \leq f \quad \forall j \in [1, m)$$

and therefore

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

After dividing both sides with A_j, we get

$$Y_j \leq f\left(1 - \frac{1}{Y_{j-1}}\right).$$

Through elementary algebra, it is known that $\forall z > 0, (1 - \frac{1}{z}) \leq \frac{z}{4}$. Therefore, we get

$$Y_j \leq \left(\frac{f}{4}\right) Y_{j-1}.$$

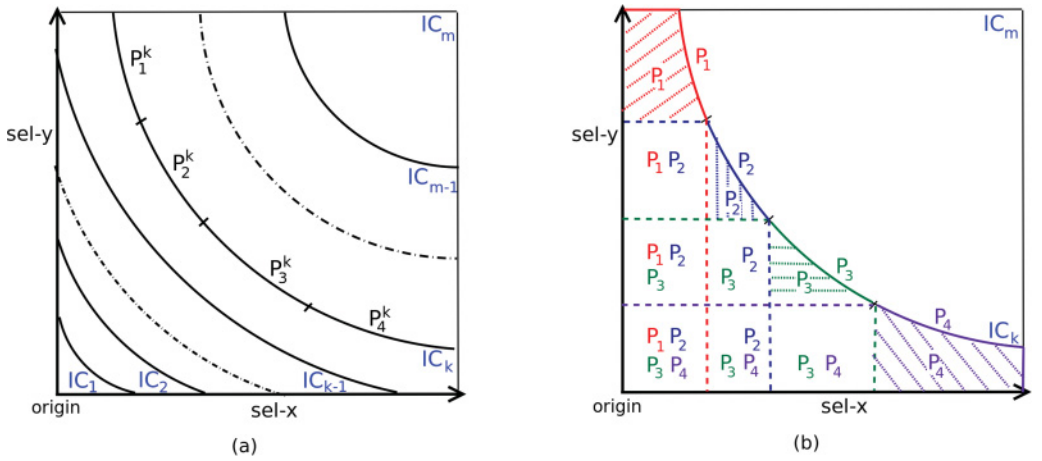

Fig. 6. 2D selectivity space: (a) isocost contours and (b) space coverage by plans on IC_k.

Since $f < 4$, it implies that the sequence Y_j is strictly decreasing with multiplicative factor < 1. With repeated application of the same inequality, we obtain

$$Y_j \leq \left(\frac{f}{4}\right)^{j-1} Y_1.$$

For sufficiently large j, this results in

$$Y_j < 1 \Rightarrow A_{j+1} < A_j,$$

which is a contradiction to our earlier observation that $A_{j+1} > A_j$. □

3.2. Multi-Dimensional Selectivity Space

We now move on to the general case of multi-dimensional selectivity error spaces. A sample 2D scenario is shown in Figure 6(a), wherein the isosurfaces 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 6(b), 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 budget $cost(IC_k)$ 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 6(b). 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 generic multi-dimensional case is shown in Algorithm 2, using the notation n_k to represent the number of plans on isosurface IC_k.

ALGORITHM 2: Multi-Dimensional Bouquet Algorithm

```
// for each isosurface IC_k
for k = 1 to m do
    // for each plan on isosurface IC_k
    for i = 1 to n_k do
        start executing bouquet plan P_i^k
        // perform cost-limited execution
        while run_cost(P_i^k) ≤ cost(IC_k) do
            execute P_i^k
            if P_i^k completes execution then
                return query result
            end
        end
        terminate P_i^k and discard partial results
    end
end
```

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

$$c_B(q_a) = \sum_{i=1}^{k} [n_i \times cost(IC_i)]. \tag{11}$$

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

$$c_B(q_a) \leq \rho \times \sum_{i=1}^{k} cost(IC_i). \tag{12}$$

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

THEOREM 3.4. *Given a query* Q *with a multidimensional* ESS*, and the associated PIC discretized with a geometric progression having common ratio r and maximum isosurface plan density* ρ*, the bouquet execution algorithm ensures that* $MSO_g = \frac{\rho r^2}{r-1}$.

Setting $r = 2$ in this expression ensures that $\text{MSO}_\text{g} = 4\rho$.

To the best of our knowledge, the above MSO bounds are the *first* such guarantees in the literature. Further, from these formulations, we can trivially infer that the ancillary metrics, ASO and MH, are bounded by MSO_g and $(\text{MSO}_\text{g} - 1)$, respectively.

4. BOUNDS ON MAXIMUM EXPECTED SUB-OPTIMALITY

Thus far, we focused on deterministic guarantees for the worst-case sub-optimality across query locations in the entire ESS. We now move on to exploring how *randomization* can be introduced in the plan bouquet algorithm, leading to guarantees on the maximum *expected* sub-optimality, that is, MESO_g. While our randomized algorithms work for arbitrary number of dimensions, for ease of presentation, we restrict our discussion here, and in the following section, to 2D ESS—hence, we will use the term *contour* to represent the isosurfaces.

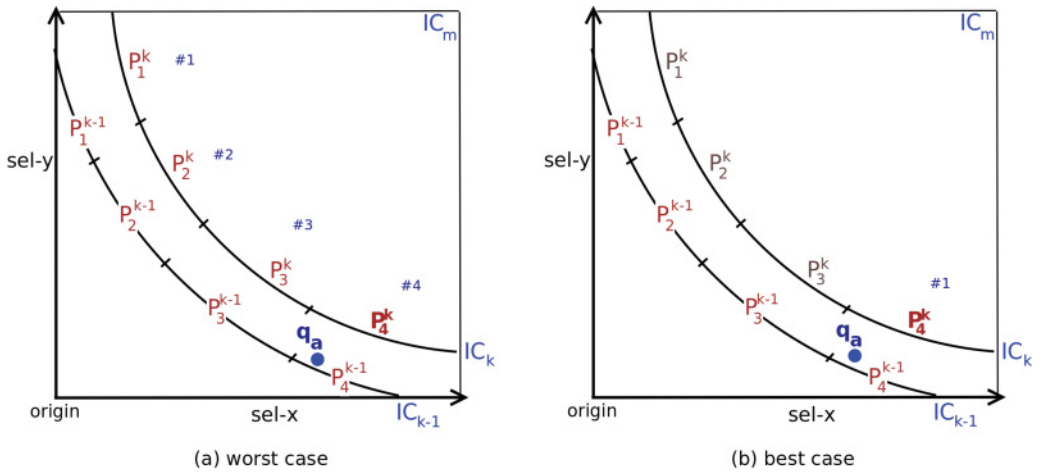

Fig. 7. Worst-case and best-case (intra-contour) plan sequences for q_a.

4.1. Randomized Intra-Contour Plan Sequence

The basic plan bouquet algorithm executes n_k plans on the k^{th} contour but does not impose any *order* on these executions. In fact, the ordering of executions has no impact on the worst-case analysis, as every plan on the contour has a region that it alone covers, suggesting exactly the same worst-case performance for any execution order.

Notwithstanding the above, the sub-optimality for a *particular* query instance could *vary* with the execution order of the contour plans. To analyze this, we split the bouquet overheads into two components: (a) the overheads suffered at the finishing contour and (b) the overheads accumulated from the earlier contours (denoted as T_B). While T_B remains the same for all query instances that lie between a consecutive pair of contours, the former is dependent on the execution order. Consider, for instance, the q_a located as shown in Figure 7(a). With the default execution order, P_1^k through P_4^k, q_a is completed by the terminal P_4^k execution, resulting in overheads of $T_B + 4 * cost(IC_k)$. On the other hand, the overheads would reduce to $T_B + cost(IC_k)$ if P_4^k was chosen as the first plan in the execution sequence (Figure 7(b)). The implication here is that the *expected* sub-optimalities can be improved by randomly choosing plan execution orders on each contour. However, minimizing this expected value may result in a weakening of the worst-case guarantee, and the tradeoff is quantified below.

We construct the following variant of the bouquet algorithm—for each contour IC_k, the execution sequence of the n_k plans is a permutation chosen uniformly at random from all possible permutations. For this variant, the performance guarantees are captured by the following result (proof in Appendix A.1):

LEMMA 4.1. *The bouquet algorithm with randomized intra-contour plan sequence provides* $MESO_g = \rho(\frac{r}{r-1} + \frac{r}{2}) + \frac{r}{2}$, *while retaining* $MSO_g = \frac{\rho r^2}{r-1}$.

Setting a common ratio of $r = 2.4$ minimizes MESO_g to $2.9\rho + 1.2$ – as a side effect, MSO_g marginally increases from 4ρ to 4.1ρ. Moreover, even if we wish to retain MSO_g of 4ρ by setting $r = 2$, then MESO_g is only mildly weakened to $3\rho + 1$. Essentially, this suggests that we can simultaneously obtain excellent performance on both metrics.

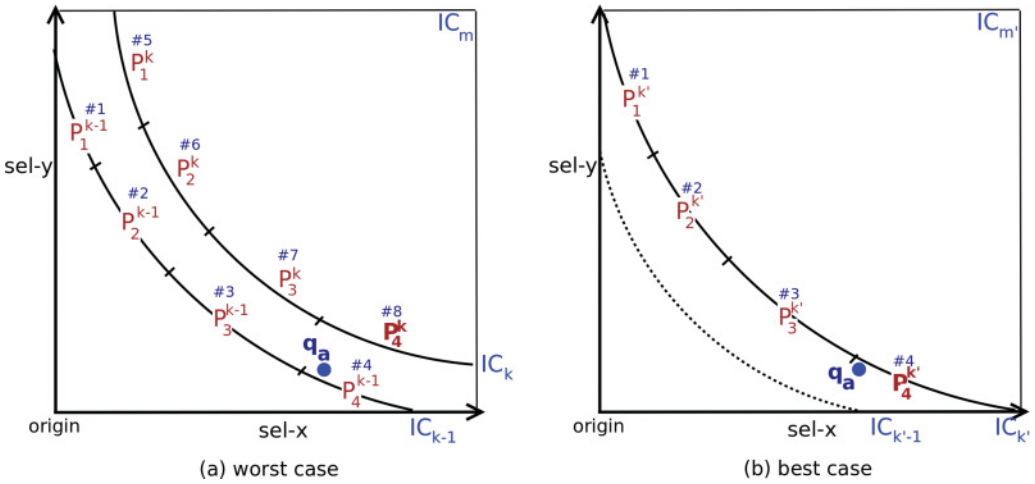

Fig. 8. Worst-case and best-case contour placements for q_a.

4.2. Randomized Contour Placement

Observe that the worst case sub-optimality instances correspond to q_a's that lie *just beyond* a contour (i.e., $c_{opt}(q_a) = cost(IC_{k-1}) + \epsilon$) since their execution finishes with a plan on the next contour, which is r-optimal. On the other hand, q_a's that lie *just below* a contour (i.e., $c_{opt}(q_a) = cost(IC_k) - \epsilon$) complete their execution with an almost-optimal plan. Such differential treatment of query instances based on their locations can be ameliorated by randomizing the *placement* of the contours—this is illustrated in Figure 8 for the example location q_a. With the original contour placement (Figure 8(a)), P_4^k completes execution for q_a expending $cost(IC_k)$, whereas after slightly repositioning the contours (Figure 8(b)), q_a is completed by $P_4^{k'}$ with $cost(IC_{k'}) \approx \frac{cost(IC_k)}{r}$.

To leverage the above, we construct a randomized variant, similar to that proposed in Lotker et al. [2008], wherein the entire geometric sequence is shifted left by a random multiplicative factor $\frac{1}{r^X}$, where X is a uniform random variable $\in [0, 1)$. That is, the cost associated with the first contour is randomized between $\frac{a}{r}$ and a, with the later contours retaining the default r cost ratio. For this variant, the performance guarantees are captured by the following result (proof in Appendix A.2):

LEMMA 4.2. *The bouquet algorithm with randomized contour placement provides* $MESO_g = \rho \frac{r}{\ln r}$, *while retaining* $MSO_g = \frac{\rho r^2}{r-1}$.

Setting a common ratio of $r = e \approx 2.72$ minimizes MESO_g to $\approx 2.72\rho$—as a side effect, MSO_g slightly increases to 4.3ρ. Moreover, even if we wish to retain the 4ρ MSO guarantee by setting $r = 2$, then MESO_g is only mildly weakened to 2.89ρ. Again, we observe simultaneous excellent performance on both metrics.

4.3. Using Both Randomization Strategies

Since the above randomization techniques are orthogonal to each other, we can also consider employing them *simultaneously*. For this variant, the performance guarantees are captured by the following result (proof in Appendix A.3):

THEOREM 4.3. *Given a query* Q *on a multi-dimensional* ESS, *the bouquet execution algorithm with contour placement randomization followed by intra-contour plan sequence randomization provides* $MESO_g = \rho \frac{(r+1)}{2\ln r} + \frac{(r-1)}{2\ln r}$, *while retaining* $MSO_g = \frac{\rho r^2}{r-1}$.

Table II. Performance of Randomized Variants of the Bouquet Algorithm

Variant	Cost-ratio(r)	$MESO_g$	MSO_g
No randomization	2	4ρ	4ρ
Randomized Plan Sequence	2	$3\rho + 1$	4ρ
	2.4	$2.9\rho + 1.2$	4.1ρ
Randomized Contour Placement	2	2.89ρ	4ρ
	2.4	2.74ρ	4.1ρ
	2.72	2.72ρ	4.3ρ
Randomized Plan Sequence & Contour Placement	2	$2.16\rho + 0.72$	4ρ
	2.4	$1.94\rho + 0.8$	4.1ρ
	2.72	$1.86\rho + 0.86$	4.3ρ
	3.6	$1.8\rho + 1$	4.98ρ

Setting $r = 3.6$ minimizes $MESO_g$ to as low as $1.8\rho+1$, while increasing MSO_g to 4.98ρ. If we wish to retain the 4ρ MSO guarantee, then $MESO_g$ increases to $2.16\rho + 0.72$.

The above results are summarized in Table II. It is noteworthy that using $r = 2$ retains MSO_g while minimizing $MESO_g$ requires different cost ratios for each variant.

With regard to implementation, the intra-contour plan sequence only requires a simple shuffling algorithm—for instance, the standard *Knuth's shuffle* [Knuth 1997]. On the other hand, randomizing the initial contour location is more complicated since, in principle, for each new starting cost, a complete rescan of the ESS is required to determine the fresh set of contours. However, even if we restricted ourselves to merely *two* instances of contour placement, corresponding to $X = 0$ and $X = 0.5$, we achieve an attractive combination of $MESO_g = 2.38\rho + 1$ and $MSO_g = 4.1\rho$, for $r = 2.4$.

5. COMPILE-TIME ENHANCEMENTS TO IMPROVE ROBUSTNESS BOUNDS

The bouquet mechanism's MSO_g guarantee of 4 for the 1D case is shown to be inherently strong in Section 3.1. However, the multi-dimensional bounds depend on ρ, the maximum plan density across the isosurfaces, which can be quite high—for instance, in excess of 150 for the 5D queries considered in our study. Therefore, to have a practically useful bound, we need to ensure that the value of ρ is reduced as far as possible.

A potential approach to achieving reduction in the "effective" value of ρ is to somehow skip some of the cost-limited executions from the original bouquet sequence. At first glance, such removal of executions may appear contrary to the principle of *exhaustive* contour execution described in Section 3. However, as we will show in the remainder of this section, it can be achieved by ensuring that the roles of skipped executions are played by carefully identified alternative executions. Specifically, we present two compile-time enhancements here for implementing such an execution skipping process.

5.1. Reducing Effective ρ with Plan Swallowing

Our first technique leverages the notion of "anorexic reduction" [Harish et al. 2007] to directly reduce the cardinality of the POSP itself. In this approach, POSP plans are allowed to "swallow" other plans, that is, occupy their regions in the ESS, if the sub-optimality introduced due to these swallowings can be bounded to a user-defined threshold, λ. Through extensive experimentation, it was shown in Harish et al. [2007] that even for complex OLAP queries with high-dimensional ESS, a λ setting 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 anorexic reduction is introduced into the plan bouquet setup, it immediately serves to steeply reduce the effective value of ρ. However, there is also a downside—the

Table III. Effect of Anorexic Reduction [$\lambda = 20\%$] on Robustness Guarantees

Error Space	ρ_{POSP}	MSO_g	ρ_{ANOREXIC}	MSO_g
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**

constant multiplication factor is increased by a factor $(1 + \lambda)$ due to the inflation in the cost budget. Overall, the deterministic guarantee is altered from $4\rho_{\text{POSP}}$ to $4(1+\lambda)\rho_{\text{ANOREXIC}}$.

Empirical evidence that this tradeoff is highly beneficial is shown in Table III, which compares for a variety of multi-dimensional error spaces, the bounds (using Equation (11)) under the original configuration and under 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 MSO_g plunges by more than an order of magnitude, going down from 379 to 30.4.

5.2. Reducing Effective ρ with Execution Covering

We now move on to describing an independent and complementary enhancement that can further reduce the effective ρ. It leverages the observation that even if a particular execution is skipped, the selectivity region covered by this execution can still be covered using execution(s) from *later* contours—of course, at a higher cost. Such skipping clearly implies an increase in sub-optimality for some individual query instances—however, from a holistic perspective, it serves to substantively reduce the effective ρ and thereby deliver much stronger MSO_g guarantees.

To formalize this enhancement, we represent the bouquet algorithm as a sequence of cost-budgeted plan executions $BS = \{E_1, E_2, \ldots, E_{terminal}\}$, where $E_{terminal}$ is the final execution that can complete all locations in the ESS. Additionally, we use the function $\phi(E_i)$ to indicate the identity of the executed plan, and $\omega(E_i)$ to represent the cost budget of the execution. So, if E_i corresponds to the execution of P_j^k, then $\phi(E_i) = P_j$ and $\omega(E_i) = cost(IC_k)$.

Now, for each E_i, denote with $R(E_i)$ the *region* of the ESS that E_i is *a priori* known to certainly complete within its budget, that is, all q s.t. $c(\phi(E_i), q) \leq \omega(E_i)$. To make this notion concrete, visual representations of $R(E_3)$ and $R(E_7)$ are shown in Figure 9, highlighted with purple horizontal lines and green slanted lines, respectively. In addition, the figure also shows that E_7 can complete all query locations in $R(E_3)$ within twice the cost budget of E_3.

For quick reference, the notations employed hereafter in this section are summarized in Table IV.

5.2.1. The Cover Relation. We define the ability of an execution to complete the ESS region of another execution as the *Cover Relation* over the set of executions. Formally, an execution E_i can *cover* execution E_j, denoted as $E_i \succeq_{cover} E_j$, if $R(E_i) \supseteq R(E_j)$. The $\succeq_{cover}$ relation imposes a partial order on the set of executions, with $E_{terminal}$ being the unique top element since $R(E_{terminal}) = \text{ESS}$.

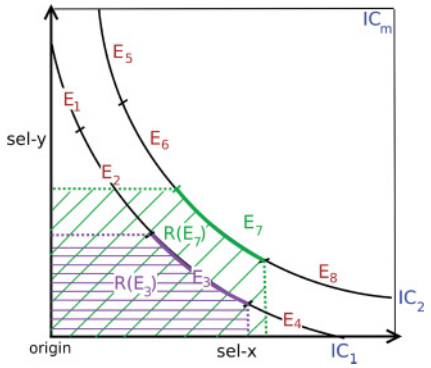

Fig. 9. E_7 can complete all locations in $R(E_3)$ with cost$(IC_2) = 2 * \text{cost}(IC_1)$.

Table IV. Reference Table for Notations in this Section

Notation	Description
$\succeq_{cover}$	Execution "cover" relation
BS	Original bouquet execution Sequence
E_i	i^{th} execution in the bouquet execution sequence
$E_{terminal}$	Final execution in the execution sequence
$\phi(E_i)$	Identity of plan used in execution E_i
$\omega(E_i)$	Cost budget of execution E_i
$R(E_i)$	Region in ESS covered by i^{th} execution
$SubOpt(E_i)$	SubOpt corresponding to i^{th} execution
CS	Covering execution Sequence
CS^k	Covering Sequence that covers the set of original executions from IC_k
CS_{opt}	Set of Optimal Covering Sequence(s)
CSI	Covering Sequence Identification Algorithm

Example. An example 2D ESS is shown in Figure 10, featuring a total of 32 executions spanning across seven contours. The corresponding *Hasse diagram* for the cover relation on the set of executions in the bouquet sequence is shown in Figure 11(a), where elements of the partial order become nodes, and non-transitive relations among the elements become edges. Further, the weight of each node is given by the cost budget of the corresponding execution, that is, $\omega(E_i)$. Since the weights are the same for all executions from a given contour, they are highlighted only once for each contour in Figure 11(a), for example, 8C for nodes 16 to 22.

5.2.2. Detailed Sub-Optimality Analysis for the Basic Bouquet Sequence. Next, we show in Figure 11(b) the detailed analysis of the basic bouquet execution sequence wrt execution cost and sub-optimality. Here, each node E_i is labeled with the accumulated overheads until and including E_i, that is $\sum_{j=1}^{i} \omega(E_j)$. While the accumulated overheads are monotonically increasing, the sub-optimality variation is not necessarily so and is given by the following recurrence formula:

$$SubOpt(E_i) = \begin{cases} SubOpt(E_{i-1}) + r & \text{if } \omega(E_i) = \omega(E_{i-1}) \\ \dfrac{SubOpt(E_{i-1})}{r} + r & \text{if } \omega(E_i) \neq \omega(E_{i-1}) \\ r & \text{if } i = 1 \end{cases}.$$

Clearly, the sub-optimality increases while working our way through a contour but may observe a dip when a contour jump happens.

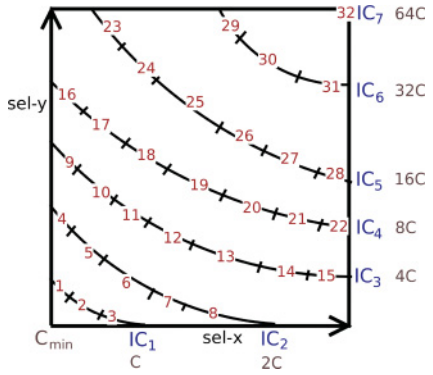

Fig. 10. Example bouquet sequence.

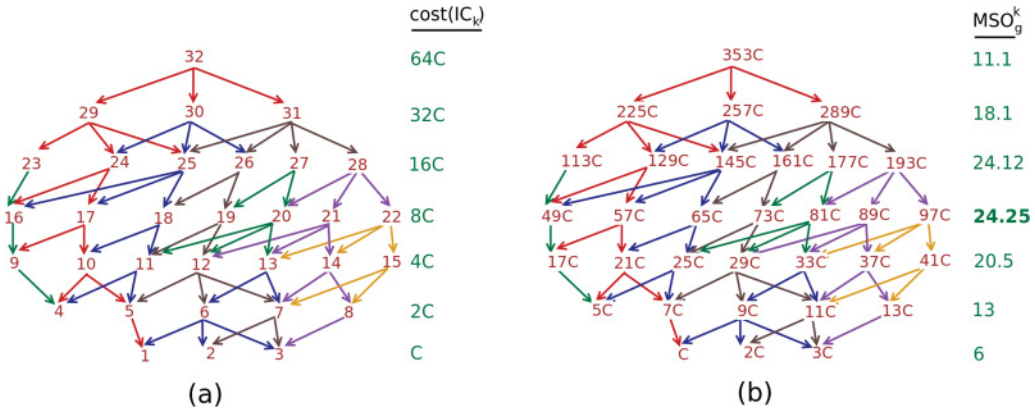

Fig. 11. (a) Hasse diagram. (b) Execution cost and sub-optimality analysis.

To elaborate further, we define MSO_g^k as the maximum sub-optimality encountered in the region between contours IC_{k-1} and IC_k and can be computed as $MSO_g^k = rn_k + \frac{MSO_g^{k-1}}{r}$, with $MSO_g^0 = 0$ for mathematical convenience. These computed values are marked besides each contour in Figure 11(b) and the overall $\text{MSO}_g = 24.25$ that occurs on the fourth contour is highlighted in boldface. It is noteworthy that, unlike the absolute bouquet overheads which increase monotonically with each new execution, MSO_g^k increases monotonically with contour k only if $n_k > \frac{MSO_g^{k-1}}{r^2}$ is satisfied for all $k \in (1, m]$.

Motivating Scenario: MSO_g Reduction Due to Execution Covering

Consider Figure 11(a), where execution E_{28} is capable of covering executions E_{20}, E_{21}, and E_{22}. That is, if E_{20}, E_{21}, E_{22} are skipped from the bouquet sequence, their associated regions $\cup_{i=20}^{22} R(E_i)$ can still be covered by execution E_{28}. Implementing this observation, as depicted in Figure 12, the effective plan density of IC_4 reduces from 7 to 6—since the cost budget of E_{28} is equivalent to two executions from IC_4. Note that this execution cover has *no impact* on sub-optimality performance until E_{19} but causes a sub-optimality reduction for all the later executions and hence reduction in MSO_g^k for all the contours beyond IC_4. Overall, MSO_g marginally reduces from 24.25 to 22.25.

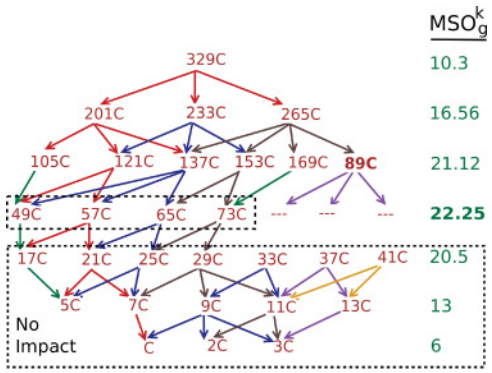

Fig. 12. Using cover $E_{28} \rightarrow \{E_{20}, E_{21}, E_{22}\}$ improves MSO_g from 24.25 to 22.25.

As a general rule, implementing a particular execution covering does not harm MSO_g if the cover's budget does not exceed the sum of the budgets of the replaced executions. That is, $E_{cover} \succeq_{cover} \{E_c, \ldots, E_{c'}\}$ can be employed if $\omega(E_{cover}) \leq \omega(E_c) + \cdots + \omega(E_{c'})$.

Covering Sequences

Extending the above single execution cover example, we define a *covering sequence* (*CS*) as an execution sequence that contains a cover for each and every execution in the original sequence. Since $E_{terminal}$ cannot be covered by any other execution, it must be present in every candidate covering sequence, implying a total of $2^{|BS|-1}$ candidates. For each candidate *CS*, MSO_g can be computed as

$$\text{MSO}_g(CS) = \max_{1 \leq a \leq m} \left[\frac{\sum_{k=1}^{a} \Omega(CS^k)}{cost(IC_{a-1})} \right],$$

where $CS^k \subset CS$ is the set of execution(s) that cover executions from IC_k and $\Omega(CS^k) = \sum_{E_i \in CS^k} \omega(E_i)$.[1] Also, $cost(IC_0) = C_{min} = \frac{cost(IC_1)}{2} + \epsilon$.

Optimal Covering Sequence. Among all the *CS* candidates, the covering sequence(s) corresponding to the minimum value of MSO_g are characterized as *optimal* covering sequences (CS_{opt}). The CS_{opt} sequences can be identified through a brute force evaluation of all candidates, but the complexity is exponential in the number of executions in the sequence, and it is therefore impractical. As a viable alternative, we propose a greedy algorithm, termed the *Covering Sequence Identification* (CSI), to find a CS with improved MSO_g. Specifically, CSI decomposes the original problem into contourwise subproblems, each of which is modeled as a *red-blue domination* problem. The subproblems are then solved efficiently using a greedy approach, and the contourwise solution nodes are stitched together to form a covering sequence (details in Appendix B.1).

As a concrete outcome of the CSI algorithm, the solution CS for the running example is shown in Figure 13(a) (the equivalent ESS coverage representation is shown in Figure 13(b)). Note that the resulting MSO_g has come down to only *14.5* as compared to 24.25 of the original bouquet sequence.

[1]If an execution is capable of acting as a cover for executions from different contours, then it is counted only once for the lowest index contour, while calculating MSO_g.

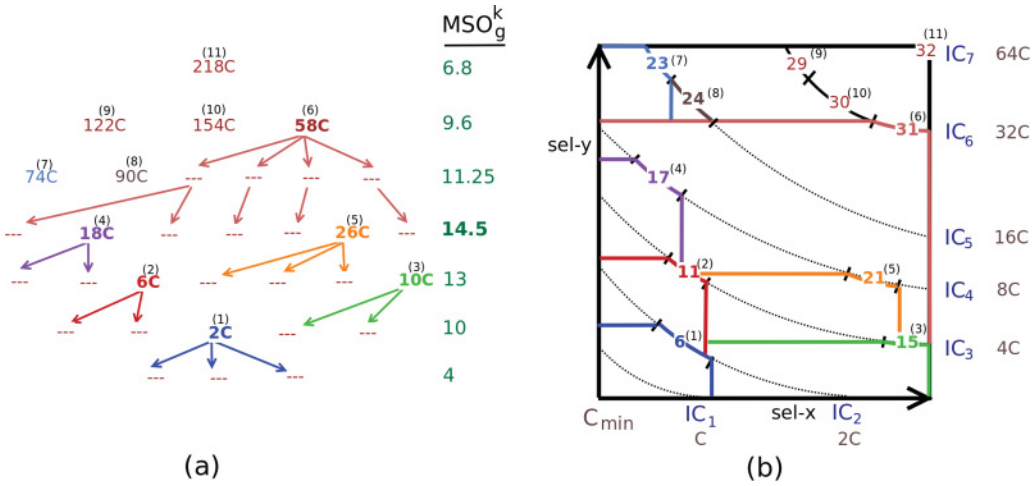

Fig. 13. (a) Execution covering sequence. (b) Modified space coverage.

With regard to the above CS solution, a few interesting sidelights emerge:

—In contrast to the original bouquet sequence, the executions are not necessarily contour-ordered in a covering sequence. For instance, E_{31} from IC_6 is executed before executions E_{23} and E_{24} from IC_5.
—The CS uses only 11 of 32 executions in the original sequence.
—The effective contour plan densities decrease from $[3, 5, 7, 7, 6, 3, 1]$ to $[2, 4, 4, 4, 2, 2, 1]$. Specifically for IC_3, ES uses seven executions from IC_3, while CS covers it employing $CS^3 = \{\underline{E_{11}, E_{15},} E_{17}, E_{21}\}$, which is equivalent to four executions (since it requires two executions, E_{17} and E_{21}, from IC_4).[2]
—An increase in execution cost (and hence sub-optimality) occurs only for the three regions covered by $E_1(C \rightarrow 2C)$, $E_4(5C \rightarrow 6C)$, and $E_9(17C \rightarrow 18C)$. For most of the remaining regions, the execution cost improves *significantly*, for example, $E_{15}(41C \rightarrow 10C)$ and $E_{31}(265C \rightarrow 58C)$. This observation suggests that along with significant reductions in MSO_g, concurrent improvements in ASO and MH may also be expected.

The effectiveness of CSI is further corroborated by its performance on queries based on the TPC-H and TPC-DS benchmarks—these results are summarized in Table V. Overall, the MSO_g never exceeded *20* even for high-dimensional queries, including those with high initial values of MSO_g. As a particularly compelling example, for a five-dimensional error-space 5D_H_Q7, the covering sequence used only 10 executions (of the original 34) and, more importantly, brought MSO_g down from 37.2 to *only 15*.

Computational Effort. Overall, the worst-case complexity for the CSI algorithm is $O(m\rho \log(m\rho))$ against $O(2^{m\rho})$ of the brute-force algorithm. Further, to be able to use this enhancement, it is required to first construct the Hasse diagram, which involves establishing the *cover* relation among all pairs of executions from consecutive contours. For this purpose, we have devised a three-step mechanism, where the proposed checks in the first two steps are computationally very cheap as compared to the third step. To elaborate, we first identify true positives by evaluating a simple necessary and sufficient criteria, then discard true negatives by evaluating another cheap to evaluate

[2]The underlined executions, E_{11} and E_{15}, are part of CS^3 but their overheads are already counted in CS^2.

Table V. Effect of Execution Covering on Robustness Guarantees

Error Space	MSO_g (Anorexic)	# Executions (Anorexic)	MSO_g (Anorexic+CSI)	# Executions (Anorexic+CSI)
3D_H_Q5	**12**	8	**8.4**	4
3D_H_Q7	**9.6**	6	**7.2**	3
4D_H_Q8	**24**	18	**15**	7
5D_H_Q7	**37.2**	34	**15**	10
3D_DS_Q15	**12**	16	**9.2**	9
3D_DS_Q96	**13**	10	**8.8**	7
4D_DS_Q7	**17.8**	14	**9.1**	7
4D_DS_Q26	**19.8**	23	**8.1**	8
4D_DS_Q91	**35.3**	34	**16**	14
5D_DS_Q19	**30.4**	24	**15**	13

necessary condition, and, finally, if the previous two steps do not prove conclusive, evaluate the computationally expensive sufficiency criteria. The complete details of this procedure can be found in Appendix B.2.

Finally, since the computational efforts required in both Hasse diagram construction and covering sequence identification depend on the number of executions in the sequence, it is recommended to use CSI only after anorexic reduction has already been employed.

6. NEXUS: ALGORITHM FOR IDENTIFYING AN ISOSURFACE

The primary inputs to the bouquet identification phase are the isosurfaces (contours in 2D) drawn on the ESS. Thus far, we had viewed each isosurface as a continuous region comprised of selectivity locations having identical cost values for their optimal plans. As a practical matter, however, we have to construct and process approximate *discretized* versions of these regions. That is, we need to use a D-dimensional grid with finite resolution *res* to approximate the ESS hypercube $[0, 1]^D$.

With this discretized ESS, an isosurface for cost C is constructed as a D-dimensional set of contiguous grid locations q such that $c_{opt}(q)$ lies in the interval $[C, (1 + \alpha)C]$, where $c_{opt}(q)$ is the cost of the optimal plan at location q and α is a tolerance factor. Since the tolerance factor could occasionally result in "thickening the surface" due to inter-surface locations also creeping into the surface set, we additionally require that each point in the surface must have at least one of its lower neighbors violating the above cost interval requirement. Finally, we assume that the resolution of the ESS grid is sufficiently high such that we can always find contiguous isocost locations even with small values of α, say, 0.05.

A straightforward strategy to identify the isosurfaces from the ESS is to first explore the discretized ESS in a exhaustive manner and then identify the locations that are acceptable for the required isocost values. But the overheads for such an approach would increase exponentially with ESS dimensionality and become impractical for typical OLAP queries. Moreover, the exhaustive enumeration is overkill for isosurface identification since: (a) we do not need information about the internal regions that lie *between* the isosurfaces and take up the vast majority of the space in ESS, and (b) we do not exploit the potential for overlapping the *identification* of later isosurfaces with the *execution* of the earlier isosurfaces, which could provide a head start in the bouquet execution process.

Motivated by the above observations, we propose in this section a *focused* approach for the identification of isosurfaces. Specifically, it quickly identifies the locations corresponding to a particular isosurface without wasting much effort on extraneous locations. For this purpose, we leverage our basic assumptions of monotonicity and

Table VI. Reference Table for Notations in this Section

Notation	Description
C	Cost of isosurface
res	Resolution of the ESS grid
α	Cost tolerance factor in isosurface identification
$c_{opt}(q)$	Optimal cost at location q in the ESS
L	A generic isosurface location in the ESS
$L_{x\pm1}$	ESS locations in immediate neighborhood of L along dimension x
S	Initial seed location for an isosurface in the ESS

smoothness of plan costs—these imply that in each dimension of the discretized ESS, the optimal costs are in increasing order and do not change abruptly.

We begin by presenting the algorithm for a 2D ESS followed by the extension to higher-dimensional selectivity spaces. The notations used in this section are summarized in Table VI.

6.1. 2D ESS

Given a location L with coordinates (x, y) in the discretized 2D ESS, we denote its three immediate "lower" neighbors as follows: $(x-1, y)$ with L_{x-1}, $(x, y-1)$ with L_{y-1}, and $(x-1, y-1)$ with L_{-1}. With these notations, the location $L(x, y)$ is included in the contour C if it satisfies the following conditions:

(a) $C \leq c_{opt}(L) \leq (1+\alpha)C$ and
(b) if $c_{opt}(L_{x-1}) > C$ and $c_{opt}(L_{y-1}) > C$ then $c_{opt}(L_{-1}) < C$.

The first condition establishes the acceptable cost interval for L, while the second ensures that at least one of L's dominated neighbors is outside of the cost interval (to prevent "surface thickening," as explained earlier).

With the above setting, the contour identification algorithm works in two phases:

(1) *Locating the Initial Seed:* Here, the aim is to find the contour location that has the *maximum "y"* coordinate and use it as a *seed* location for the next phase of the algorithm. This extreme point can only lie on either the left edge or the top edge of the ESS, that is, (0,0) to (0, res) or (0,res) to (res, res). To determine the correct edge, we simply cost these three ESS corners and determine which edge includes C in its range of values. Once the edge has been determined, the exact location S, to serve as the initial seed, is determined using a *binary search* on that edge.
(2) *Neighborhood EXploration Using Seed (NEXUS):* Since the seed has the maximum "y" location, for locating our next isocost point, we need to only consider the third and fourth quadrants relative to the seed as origin. However, locations in the third quadrant are already known to be *unacceptable* due to PCM. Therefore, the initial seed location S can be used to recursively generate new seed locations solely in the fourth quadrant and thus grow the contour.

   For a given seed location $S(x, y)$, we denote the location $(x+1, y)$ with S_{x+1} and the location $(x, y-1)$ with S_{y-1}. By virtue of PCM, we know that $c_{opt}(S_{x+1}) > c_{opt}(S)$ and $c_{opt}(S_{y-1}) < c_{opt}(S)$. We find from the query optimizer the optimal costs for these candidate seed locations and choose the new seed based on the following simple criterion:

   If $c_{opt}(S_{y-1}) < C$, then set $S = S_{x+1}$ else $S = S_{y-1}$.

   The end of this recursive routine is marked by the non-existence of both S_{x+1} and S_{y-1} in the ESS grid.

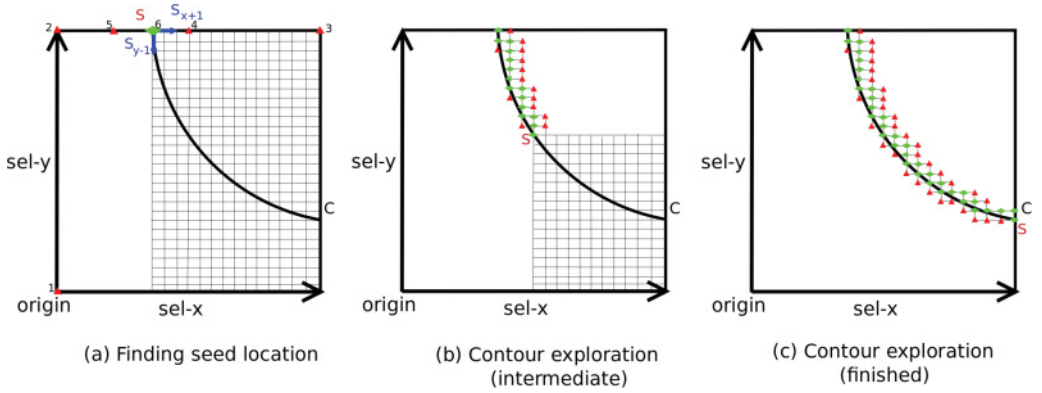

Fig. 14. Example contour exploration in 2D ESS.

A sample working of the above algorithm is visually demonstrated in Figure 14. First, the identification of the initial seed S using six optimization calls is shown in Figure 14(a). Then the recursive contour exploration in the fourth quadrant of S is shown in Figure 14(b); here, the optimized locations are marked with either a red triangle ▲ or a green dot •. The triangles indicate locations that were explored but rejected, whereas the dots constitute the accepted contour locations. Finally, Figure 14(c) shows the contour exploration completing when S hits the ESS boundary.

Discussion. It is noteworthy that $\#red▲ = \#green•$, that is, the algorithm performs exactly *twice* the number of optimizer calls as compared to the optimal algorithm that finds only acceptable contour locations. This is because, at any point during contour exploration, there are *exactly two* candidates for the new seed, S_{x+1} and S_{y-1}, and one of them will definitely be on the (accepted) contour. In fact, it is easy to see that since the decision is based solely on S_{y-1}, the optimization call for S_{x+1} should be invoked only if required and thereby further reduce the number of wasted optimization calls.

6.2. Extension to nD ESS

Next, we show that the neighborhood exploration approach for contour identification can be easily extended to general multi-dimensional ESS. For this purpose, we start with the extended algorithm for 3D ESS that systematically invokes different instances of the 2D algorithm.

Locating the Initial 3D Seed. Here, the initial seed S is the isosurface location with the maximum z coordinate. To find this point, it is first checked whether the seed lies on the edge $(0, 0, 0)$ to $(0, 0, res)$, which implies that $S = (0, 0, z)$ with $z < res$. If yes, then the seed can be determined by using a binary search on this edge—this corresponds to Case 1 in Figure 15. If no, then the initial seed is located using a procedure similar to 2D ESS for the XY slice with $z = res$, which is visualized as Case 2 in Figure 15; here, there are two possibilities: $S = (0, y, res)$ with $y < res$ (Case 2a) or $S = (x, res, res)$ with $x < res$ (Case 2b).

3D Isosurface Exploration. We first explain the isosurface exploration phase for Case 2b. To identify all isosurface locations with $z = res$, we use the 2D exploration algorithm for the XY slice with $z = res$ and grow the initial seed S as explained previously. For exploring the locations with lower values of z, the initial seeds for each XY slice are generated by 2D exploration of the XZ slice corresponding to $y = res$, using the initial seed S and candidate locations S_{x+1} and S_{z-1}.

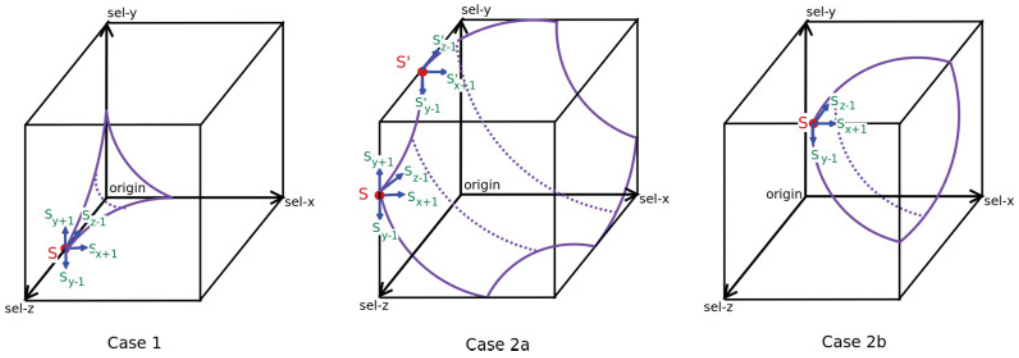

Fig. 15. Example isosurface exploration in 3D ESS.

Similarly, in Case 1, the initial seeds for each lower value of z are generated by exploring the YZ slice corresponding to $x = 0$, starting with an initial seed S and candidate locations S_{y+1} and S_{z-1}. Finally, the algorithm for Case 2a proceeds in two sub-phases where the first sub-phase is similar to Case 1 until it finds a seed with $y = res$ (shown as S' in Case 2a of Figure 15), and thereafter, in the second phase, it follows an algorithm similar to Case 2b.

Generic nD Algorithm. In D-dimensional space, the initial seed location is of the form $(0^s, v, res^t)$ where $0 < v < res$ and $s + t = D - 1$. Given such a seed, the dimension pair (d_{s+1}, d_{s+1+t}) is used to generate more seeds through the 2D algorithm, and for each such seed, the $D - 1$-dimensional subproblem over the dimensions $(d_1, d_2, \ldots, d_{s+t})$ is recursively solved. The recursion terminates with the completion of 2D exploration of the dimension pair (d_{s+1}, d_{s+1+t}).

6.3. Impact on Bouquet Identification Overheads

Overall, NEXUS can be used to either: (a) enable an early start for the bouquet execution phase without invoking CSI or, alternatively, (b) reduce the total effort of identifying all isosurface plans before using CSI (by ignoring the ESS regions that lie in between the isosurfaces). In addition, this approach also makes isosurface exploration a highly parallelizable task since, in principle, a new thread can be created whenever a seed is generated for a lower-dimensional subspace.

7. IMPLEMENTATION DETAILS

Given a user query Q, the first step is to identify the error-prone selectivity dimensions in the query. For this, we can leverage the approach proposed in Kabra and DeWitt [1998], 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 [0,1] selectivity 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 row cardinality.

Once the ESS is finalized, the query execution workflow of the bouquet approach becomes operational, as shown in Figure 16. For this purpose, the database engine needs to support the following functionalities: (1) selectivity injection, (2) abstract

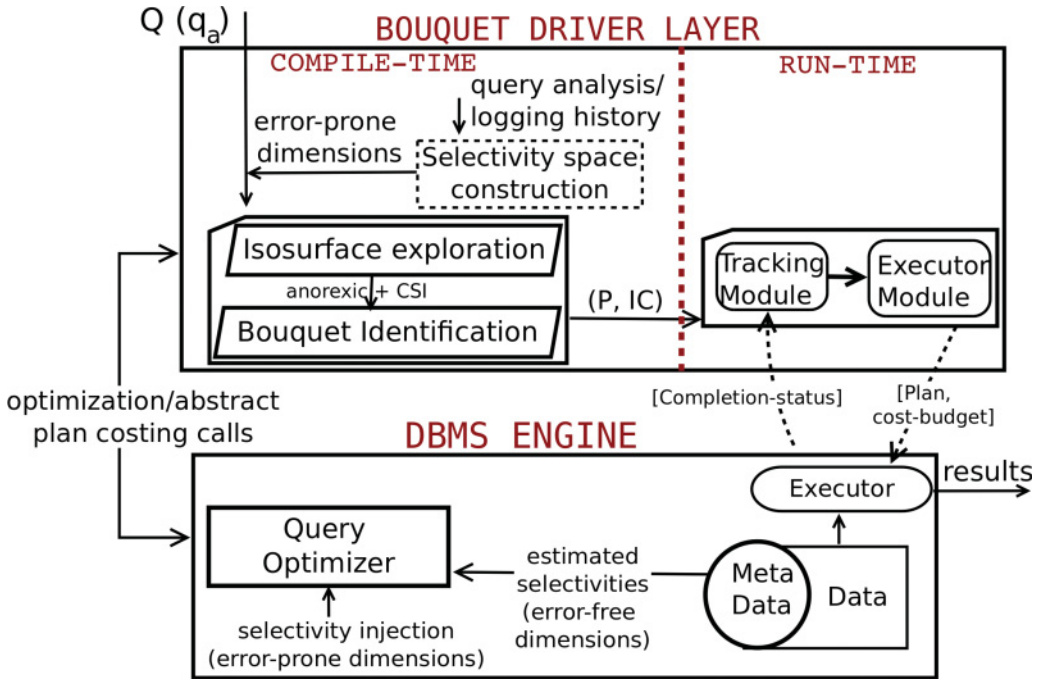

Fig. 16. Architecture of bouquet mechanism.

plan costing and execution, and (3) cost-limited partial execution of plans. Next, we elaborate on the usage of each of these features, followed by implementation details of the bouquet driver layer.

7.1. Selectivity Injection

For isosurface exploration using the algorithm described in Section 6, 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 highly 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 [IBM 2003]).

7.2. Abstract Plan Costing and Execution

Once the isosurfaces have been explored, we need to reduce their plan densities using the anorexic reduction technique, as explained in Section 5.1. This is achieved through the FPC variant of the Cost Greedy algorithm [Harish et al. 2007], which requires an abstract plan costing feature for estimating the cost of a plan outside its optimality region. This feature is already supported by some commercial optimizers (e.g., SQLSERVER [2010]).

Further, during the bouquet execution phase, we need to be able to instruct the execution engine to execute a particular bouquet plan. This feature also is currently provided by a few commercial systems (e.g., SQLSERVER [2010]).

7.3. Cost-Limited Execution

The bouquet approach requires, in principle, only a simple "timer" that keeps track of elapsed time and terminates plan executions if they exceed their assigned cost budgets. No material changes need to be made in the engine internals to support this feature. The premature termination of plans can be achieved easily using the *statement.cancel()* functionality supported by JDBC drivers. Note that, although bouquet identification provides budgets in terms of abstract optimizer cost units, they can be converted to equivalent time budgets through the techniques proposed in Wu et al. [2013].

7.4. Bouquet Driver Layer

As highlighted in Figure 16, we have an external program, the "Bouquet Driver," which treats the query optimizer and executor as black boxes. First, it interacts with the query optimizer module to determine the isosurfaces and the plan bouquet. It then performs executions of the bouquet plans using an *execution* client and a *tracking* client. The execution client selects the plan to be executed next, while the tracking client keeps track of the time elapsed and terminates the execution if the allotted time budget is exhausted.

8. 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 describe experiments that show: (a) spatial distribution of robustness in the ESS, (b) low bouquet cardinalities, (c) low sensitivity of the MSO_g to the λ reduction parameter, and (d) extension of the results to commercial databases. As specified in Section 2, the entire evaluation is carried out using optimizer costs, while assuming that all combinations of the actual and estimated query locations are possible in the ESS.

Before going into the evaluation details, we describe the experimental setup and the rationale behind the choice of comparative techniques. This is followed by a brief discussion on the compile-time overheads incurred by the bouquet algorithm.

8.1. Experimental Setup

Database Environment. The test queries are chosen from the TPC-H and TPC-DS benchmarks to cover a spectrum of join-graph geometries, including *chain*, *star*, *branch*, and so on, 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. 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 VII—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.

System Environment. For the most part, the database engine used in our experiments is PostgreSQL [2009], equipped with the API features described in Section 7. Specifically, the first two features were introduced with minimal changes to the source code. On the other hand, cost-budgeted execution is natively supported by invoking the following command at the tracking client: "*select pg_cancel_backend(process_id)*." The

Table VII. Query Workload Specifications

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	4D_DS_Q26	star(5)	341
5D_H_Q7	chain(6)	50	4D_DS_Q91	branch(7)	149
3D_DS_Q15	chain(4)	668	5D_DS_Q19	branch(6)	183

required *process_id* (of the execution client) can be found in the view *pg_stat_activity*, which is maintained by the engine itself.

The hardware platform is a vanilla Sun Ultra 24 workstation with 8 GB memory and 1.2 TB of hard disk.

Comparative Techniques. In the remainder of this section, we compare the bouquet algorithm (with anorexic parameter $\lambda = 20\%$ and CSI enhancements) against the native PostgreSQL optimizer and the SEER robust plan selection algorithm [Harish et al. 2008].

SEER uses a mathematical model of plan cost behavior in conjunction with anorexic reduction to provide replacement plans ($P_{rep}(q_e)$ for q_e) that, at all locations in the ESS, either improve on the native optimizer's performance or are worse by at most the λ factor. It is important to note here that, in the SEER framework, the comparative yardstick is $P_{opt}(q_e)$, the optimal plan at the *estimated* location, whereas in our work, the comparison is with $P_{opt}(q_a)$, the optimal plan at the *actual* location. Still, it has been shown in Harish et al. [2008] that in many cases $c(P_{rep}(q_e), q_a) \ll c(P_{opt}(q_e), q_a)$, and, hence, SEER is expected to perform better than the native optimizer on our sub-optimality-based metrics. Finally, since $c(P_{rep}(q_e), q_a) \leq (1 + \lambda) \times c(P_{opt}(q_e), q_a) \quad \forall q_a \in ESS$, we can infer that MH $\leq \lambda$ with SEER.

On the other hand, purely heuristic-based reoptimization techniques, such as POP Markl et al. [2004] and Rio Babu et al. [2005], are not included in the evaluation suite. No doubt they can be very effective when the estimation errors are small in magnitude and number. But when the errors are significant, as is commonplace in practice [Lohman 2014], their performance on our metrics (MSO or MH) could be *arbitrarily poor*. This is because of their inability to provide worst-case performance guarantees—in fact, they are unable to do so with regard to both $P_{opt}(q_e)$ and $P_{opt}(q_a)$, as explained in detail in Dutt and Haritsa [2014b]).

Further, the heuristics that POP and Rio employ are more appropriate for low-dimensional spaces—for example, that near-optimality of a plan at the corners of the principal diagonal of the error space implies near-optimality in the interior of this space or that the selectivity validity ranges found by comparing only with the class of structure-equivalent plans provide good approximations to the true ranges. Therefore, these techniques may not work well when faced with large multi-dimensional estimation errors, which is the primary target of our work.

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.

8.2. Compile-Time Overheads

The computationally expensive aspect of BOU's compile-time phase is the identification of the plans on the isosurfaces of the ESS. For this task, we have proposed a new algorithm NEXUS, as described in Section 6, that can selectively explore locations for a particular isosurface, and ignore the remaining portion of the ESS. Currently, using

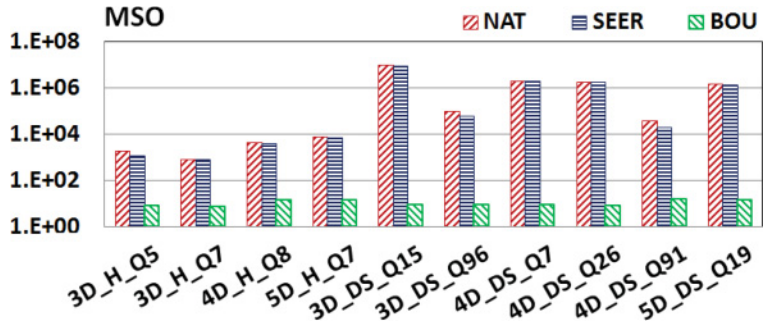

Fig. 17. Empirical MSO Performance.

only *single-core* processing, the compile-time overheads for 3D, 4D, and 5D queries with NEXUS are typically a few minutes, a few hours, and several hours, respectively. Although the overheads continue to increase exponentially with the number of ESS dimensions, we wish to highlight that NEXUS is highly parallelizable and can therefore exploit modern multi-core architectures to substantively ameliorate, in absolute terms, these overheads.

Further, as mentioned in Section 6, the plan bouquet execution can be overlapped with its compilation—specifically, execution can be started as soon as the first plan on the first isosurface is identified, and the identification of subsequent plans can be carried out concurrently with the ongoing executions.

Finally, note that the isosurface identification process is a one-time exercise, and its overhead can be amortized by repeated invocations of the same query, which often happens with "canned" form-based interfaces in the enterprise domain.

8.3. Empirical Worst-Case Performance (MSO)

In Figure 17, the empirical 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. Second, 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 *fewer than 10* across all the queries.

8.4. 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 18, 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 5 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 ESS that makes it extremely difficult to find universally safe replacements that are also substantively beneficial.

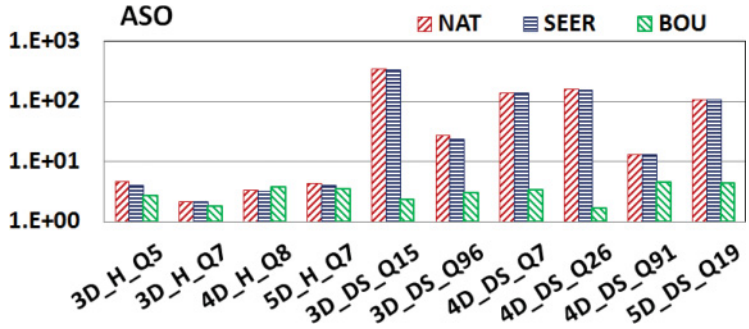

Fig. 18. ASO performance.

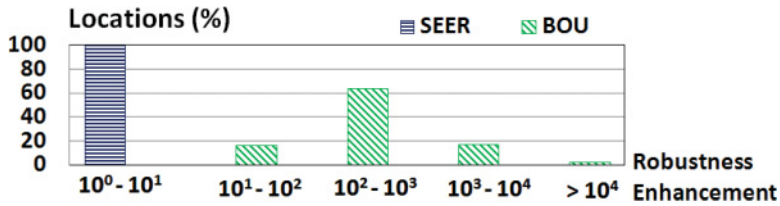

Fig. 19. Distribution of enhanced robustness (5D_DS_Q19).

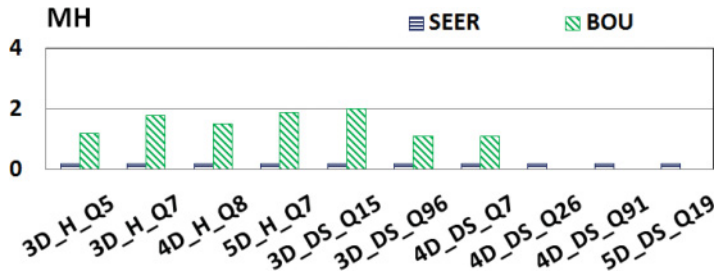

Fig. 20. MaxHarm performance.

8.5. 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 19, where we find that for the vast majority of locations (close to 85%), 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.

8.6. 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, that is, $SubOpt(*, q_a) > SubOpt_{worst}(q_a)$. This aspect is quantified in Figure 20, where the maximum harm is shown (on a linear scale) for our query test suite. We observe that BOU may be up to a factor of 2 worse than NAT. Moreover, SEER now steals a march over BOU since it *guarantees* that MH never exceeds λ (= 0.2).

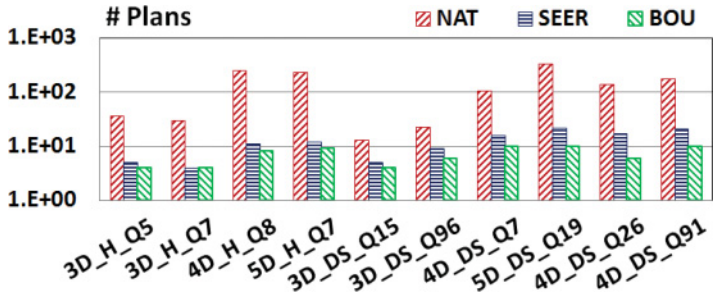

Fig. 21. Bouquet cardinality.

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. Further, even in these few cases the harm is limited ($\leq$ MSO-1), especially when viewed against the order of magnitude improvements achieved in the beneficial scenarios.

8.7. Plan Cardinalities

The plan cardinalities of NAT, SEER, and BOU are shown on a log-scale in Figure 21. 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 the isosurfaces. 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*.

8.8. MSO_g Sensitivity to λ Setting

Until now, both BOU and SEER have been evaluated empirically over different metrics by setting the reduction-parameter λ to 20%, a value that had been found in Harish et al. 2007] 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 22, the MSO_g 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_g values drop steeply with the use of covering enhancement and improve even further when 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.

8.9. 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 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 23, we find that here also large values of MSO and ASO are obtained for NAT and SEER. Further, BOU continues to

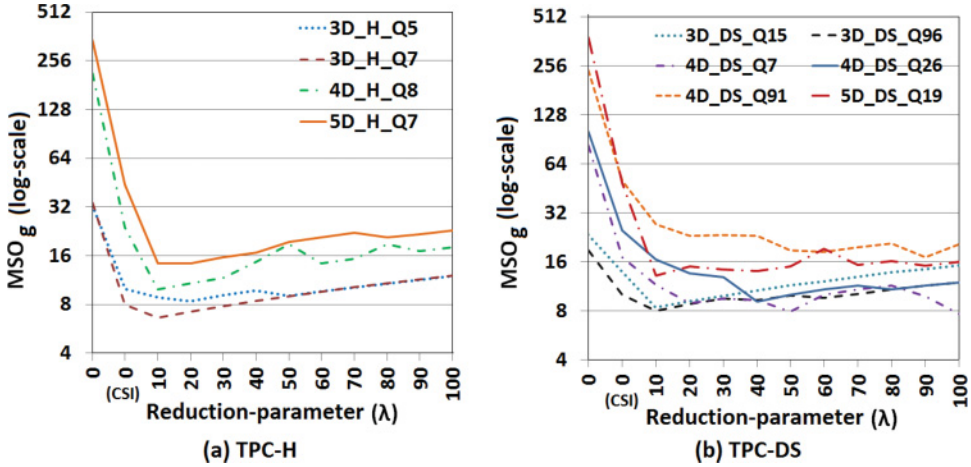

Fig. 22. MSO_g vs reduction-parameter (λ).

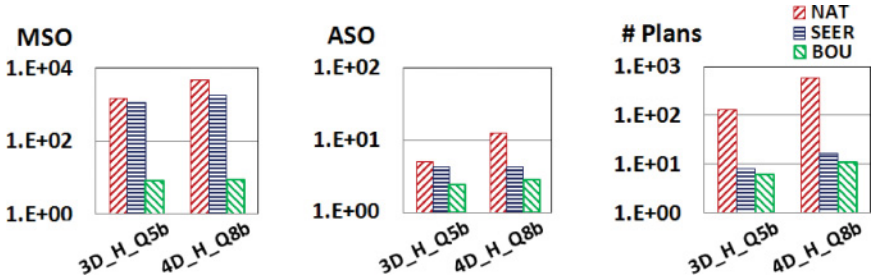

Fig. 23. Commercial engine performance (log-scale).

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.

9. RELATED WORK

A rich body of literature is available pertaining to selectivity estimation issues [Deshpande et al. 2007]. We start with the overview of the closely related techniques that can be collectively termed as *plan-switching approaches*, as they involve runtime switching among complete query plans. At first glance, our bouquet approach, with its partial execution of multiple plans, may appear very similar to runtime re-optimization techniques such as POP [Markl et al. 2004] and Rio [Babu et al. 2005]. However, there are key differences: First, they start with the optimizer's estimate as the initial seed and then conduct a full-scale re-optimization if the estimate is 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.

Second, 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 selectivity 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 based solely on the performance at the *corners* of the ESS has not been justified.

Recently, a novel interleaved optimization and execution approach was proposed in Neumann and Galindo-Legaria [2013], 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 Moerkotte et al. [2009] that minimizes the multiplicative error. While this technique substantively reduces the execution overheads, it also provides no guarantees as it is largely based on heuristics.

Techniques that use a single plan during the entire query execution [Chu et al. 2002; Babcock and Chaudhuri 2005; Harish et al. 2008; Moerkotte et al. 2009; Chaudhuri et al. 2010; Tzoumas et al. 2013] 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" [Avnur and Hellerstein 2000] and "plan per tuple group" [Polyzotis 2005].

Interestingly, a recent proposal [Ngo et al. 2012] that decides the join strategy on a *per-tuple* basis also provides performance guarantees but in terms of the *query output size*. Moreover, these guarantees are predicated on the presence of a futuristic join operator (similar to *Leapfrog triejoin* [Veldhuizen 2014]) that can concurrently combine several relations. In contrast, we provide execution time sub-optimality guarantees for the conventional query processing frameworks that are prevalent in current RDBMS engines, wherein typically only binary joins are supported.

Our technique may also superficially look similar to PQO techniques (e.g., PPQO [Bizarro et al. 2009]), 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 runtime. This is a major difference from "plan-morphing" approaches, where the execution plan may be substantially modified at runtime using custom-designed operators, for example, *chooseplan* [Cole and Graefe 1994], *switch* [Babu et al. 2005], and *feedback* [Chaudhuri et al. 2008].

Another direction to handle wrong plan choices has been to invent *adaptive operators* in the query plan—for instance, scan in Borovica-Gajic et al. [2015] and join in Graefe [2012]. Although they have been shown to be quite effective in cases when the sub-optimality of the plan is a result of wrong operator decisions, not much progress has been made for cases when the sub-optimality is caused due to wrong choice of join-order itself.

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 [Babu et al. 2005; Harish et al. 2008; Kabra and DeWitt 1998; Markl et al. 2004; Neumann and Galindo-Legaria 2013; Tzoumas et al. 2013], (b) improved average performance and/or reduced variance [Chu et al. 2002; Chaudhuri et al. 2010; Babcock and Chaudhuri 2005], (c) improved accuracy of

selectivity estimation structures [Aboulnaga and Chaudhuri 1999], (d) bounded impact of multiplicative estimation errors [Moerkotte et al. 2009], and (e) smooth performance degradation [Graefe 2012; Borovica-Gajic et al. 2015].

We introduced the notion of worst case performance guarantees using the plan bouquet algorithm in Dutt and Haritsa [2014a] and also investigated a variety of compile-time and runtime enhancements. Our current work has the following additional contributions: (a) Proposes *randomized* variants of the basic plan bouquet algorithm and guarantees on maximum expected sub-optimality in Section 4; (b) introduces another potent compile-time enhancement, *execution covering sequence*, that significantly lowers the *effective* isosurface plan densities, resulting in materially improved worst case guarantees, in Section 5.2; (c) presents a completely new mechanism for identification of isosurfaces in Section 6, that dramatically reduces the large compile-time overheads associated with bouquet identification; and (d) demonstrates how the plan bouquet approach can be successfully implemented in a completely *non-invasive* manner, leveraging existing database engine API functionalities, while retaining its performance profile, in Sections 7 and 8.

10. CRITIQUE OF THE 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 *a priori* known to be small, re-optimization techniques such as those by Markl et al. [2004] and Babu et al. [2005], 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 *a priori* 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: First, they are poor at serving *latency-sensitive* applications as they have to perforce wait for the final plan execution to return result tuples. Second, 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 can 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 runtime. Notwithstanding, these plan-switching techniques are now featured even in commercial products (e.g. Oracle [2013]).

There are also a few problems that are *specific* to the bouquet approach: First, 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.

Second, the dimensionality of the error space can be large for complex queries, which has direct impact on the bouquet identification overheads as well as MSO_g. This problem is exacerbated by the presence of parameterized predicates, which need to be included as dimensions in the ESS, in addition to the error-prone selectivity predicates. However, at the same time, it is also important to note that 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 and (ii) the join selectivities for PK-FK joins can be estimated accurately if the entire PK relation participates in the join.

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.

11. CONCLUSIONS

Selectivity estimation errors resulting in poor query processing performance are part of the database folklore. In this article, we investigated a new approach to this classical problem, wherein the estimation process was completely discarded for error-prone predicates. Instead, such selectivities were progressively discovered at runtime through a carefully graded sequence of cost-budgeted executions from a "plan bouquet." The execution sequence, which followed a cost-doubling geometric progression, ensured that the overheads are bounded, thereby ensuring MSO_g of 4 times the plan cardinality of the densest isosurface. Also, incorporating randomized strategies in the above algorithm brought down the multiple of 4 to only 1.8 as the guarantee on the expected performance. To the best of our knowledge, such bounds have not been previously presented in the database literature.

We also proposed an efficient isosurface identification algorithm for pragmatic overheads during bouquet identification and two compile time enhancements that significantly improved the worst-case guarantees. Together they ensured that MSO_g was less than 20 across all the queries in our evaluation set, an enormous improvement compared to the MSO performance of the native optimizer, wherein this metric ranged from the thousands to the millions. Further, the bouquet's ASO performance was always either comparable to or much better than the native optimizer, with most of the query locations having a sub-optimality of less than 4. 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 2 or less.

Since the bouquet technique works best when the cost model is perfect, a potential future work is to extend the tuning techniques of Wu et al. [2013] to further reduce the cost modeling errors. Developing incremental bouquet maintenance techniques is another interesting research challenge, as also mechanisms for reducing the dimensionality of the ESS space to only those that materially impact the robustness guarantees.

In closing, the bouquet approach promises an easy-to-deploy solution with guaranteed performance and repeatability in query execution, features that had hitherto not been available, thereby opening up new possibilities for robust query processing.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank Neeldhara Misra, C. Rajmohan and Bruhathi Sundarmurthy for stimulating discussions about different parts of this work. We also thank Prasad Deshpande for helpful feedback on the draft version of this article.

REFERENCES

Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning histograms: Building histograms without looking at data. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'99)*. 181–192.

Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'00)*. 261–272.

Brian Babcock and Surajit Chaudhuri. 2005. Towards a robust query optimizer: A principled and practical approach. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'05)*. 119–130.

Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'05)*. 107–118.

Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive parametric query optimization. *IEEE Trans. Knowl. Data Eng.* 21, 4 (2009), 582–594.

Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2015. Smooth scan: Statistics-oblivious access paths. In *Proc. of the 31st IEEE Intl. Conf. on Data Engg. (ICDE'15)*. 315–326.

Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. 2010. Variance aware optimization of parameterized queries. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'10)*. 531–542.

Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2008. A pay-as-you-go framework for query execution feedback. *In Proc. VLDB* 1, 1 (2008), 1141–1152.

Marek Chrobak, Claire Kenyon, John Noga, and Neal E. Young. 2008. Incremental medians via online bidding. *Algorithmica* 50, 4 (2008), 455–478.

Francis Chu, Joseph Halpern, and Johannes Gehrke. 2002. Least expected cost query optimization: What can we expect? In *Proc. of the 21st Symposium on Principles of Database Systems (PODS'02)*. 293–302.

Richard L. Cole and Goetz Graefe. 1994. Optimization of dynamic query evaluation plans. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'94)*. 150–160.

Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Foundations and Trends in Databases* 1, 1 (2007), 1–140.

Anshuman Dutt and Jayant R. Haritsa. 2014a. Plan bouquets: Query processing without selectivity estimation. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'14)*. 1039–1050.

Anshuman Dutt and Jayant R. Haritsa. 2014b. *Query Processing without Estimation*. Technical Report TR-2014-01. Database Systems Lab, SERC/CSA, Indian Institute of Science. Retrieved from http://dsl.serc.iisc.ernet.in/publications/TR/TR-2014-01.pdf.

Fedor V. Fomin and Alexey A. Stepanov. 2007. Counting minimum weighted dominating sets. In *Computing and Combinatorics*. Lecture Notes in Computer Science, Vol. 4598. Springer, Berlin, 165–175.

Goetz Graefe. 2012. New algorithms for join and grouping operations. *Comput. Sci.* 27, 1 (Feb. 2012), 3–27.

Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. 2012. Robust query processing (Dagstuhl seminar 12321). *Dagstuhl Rep.* 2, 8 (2012), 1–15.

D. Harish, Pooja N. Darera, and Jayant R. Haritsa. 2007. On the production of anorexic plan diagrams. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB'07)*. 1081–1092.

D. Harish, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *In Proc. VLDB* 1, 1 (2008), 1124–1140.

IBM. 2003. Using a SELECTIVITY clause to influence the optimizer. Retrieved from www.ibm.com/developerworks/data/library/tips/dm-0312yip/.

Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the propagation of errors in the size of join results. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'91)*. 268–277.

Navin Kabra and David J. DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'98)*. 106–117.

Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Professional, Boston, MA, 145–146.

Guy Lohman. 2014. Is Query Optimization a Solved Problem? Retrieved from http://wp.sigmod.org/?author=20.

Zvi Lotker, Boaz Patt-Shamir, and Dror Rawitz. 2008. Rent, lease or buy: Randomized algorithms for multislope ski rental. In *Proc. of the 25th Annual Symposium on Theoretical Aspects of Computer Science.* 503–514.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'04)*. 659–670.

Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *In Proc. VLDB* 2, 1 (Aug. 2009), 982–993.

Thomas Neumann and César A. Galindo-Legaria. 2013. Taking the edge off cardinality estimation errors using incremental execution. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*. 73–92.

Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [Extended abstract]. In *Proc. of the 31st Symposium on Principles of Database Systems (PODS'12)*. 37–48.

Oracle. 2013. Optimizer with Oracle Database 12c. Retrieved from www.oracle.com/ocom/groups/public/@otn/documents/webcontent/1963236.pdf.

Neoklis Polyzotis. 2005. Selectivity-based partitioning: A divide-and-union paradigm for effective query optimization. In *Proc. of the 14th ACM Intl. Conf. on Information and Knowledge Management (CIKM'05)*. ACM, New York, NY, 720–727.

PostgreSQL. 2009. PostgreSQL 8.4. www.postgresql.org/docs/8.4/static/release.html. (2009).

Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *Proc. of the 31st Intl. Conf. Very Large Data Bases (VLDB'05)*. 1228–1239.

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'79)*. 23–34.

SQLSERVER. 2010. Using the USE PLAN Query Hint. Retrieved from technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx.

Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's learning optimizer. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases (VLDB'01)*. 19–28.

Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2013. Efficiently adapting graphical models for selectivity estimation. *VLDB J.* 22, 1 (2013), 3–27.

Todd L. Veldhuizen. 2014. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. of the 17th Intl. Conf. on Database Theory (ICDT'14)*. 96–106.

Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable? In *Proc. of the 29th IEEE Intl. Conf. on Data Engg. (ICDE'13)*. 1081–1092.

Received March 2015; revised September 2015; accepted January 2016