A Concave Path to Low-overhead Robust Query Processing

Srinivas Karthik   Jayant R. Haritsa
Indian Institute of Science, Bangalore, India
{srinivasv,haritsa}@iisc.ac.in

Sreyash Kenkre   Vinayaka Pandit
IBM Research, Bangalore, India
{srekenkr,pvinayak}@in.ibm.com

ABSTRACT

To address the classical selectivity estimation problem in database systems, a radically different query processing technique called `PlanBouquet` was proposed in 2014. In this approach, the estimation process is completely abandoned and replaced with a calibrated selectivity discovery mechanism. The beneficial outcome is that provable guarantees are obtained on worst-case execution performance, thereby facilitating robust query processing. An improved version of `PlanBouquet`, called `SpillBound` (SB), which significantly accelerates the selectivity discovery process, and provides platform-independent performance guarantees, was presented two years ago.

Notwithstanding its benefits, a limitation of `SpillBound` is that its guarantees are predicated on expending enormous pre-processing efforts during query compilation, making it suitable only for canned queries that are invoked repeatedly. In this paper, we address this limitation by leveraging the fact that plan cost functions typically exhibit *concave down behavior* with regard to predicate selectivities. Specifically, we design `FrugalSpillBound`, which provably achieves extremely attractive tradeoffs between the performance guarantees and the compilation overheads. For instance, relaxing the performance guarantee by a factor of two typically results in at least *two orders of magnitude* reduction in the overheads. Further, when empirically evaluated on benchmark OLAP queries, the decrease in overheads is even greater, often more than *three* orders of magnitude. Therefore, `FrugalSpillBound` substantively extends robust query processing towards supporting ad-hoc queries.

1. INTRODUCTION

The traditional approaches for optimizing declarative OLAP queries (e.g. [20, 7]) are contingent on estimating a host of *predicate selectivities* in order to find the optimal execution plan. For in-

stance, even the relatively simple TPC-H query shown in Figure 1, which lists the order dates for cheap parts, requires as many as *four* selectivities to be estimated (corresponding to the filter, projection and pair of join predicates).

---

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

Figure 1: Example TPC-H Query

Unfortunately, in practice, these selectivity estimates are often significantly in error with respect to the actual selectivities encountered during query execution – to the extent that *orders of magnitude* errors have been routinely reported in the literature [1, 14, 16]. These estimation errors cumulatively result in highly sub-optimal choices of execution plans, and corresponding blowups in query response times. For instance, when Query 19 of the TPC-DS benchmark is executed on contemporary database engines, the worst-case slowdown, relative to a hypothetical oracle that magically knows the correct selectivities, can exceed a *million*! [5]

SpillBound. To address the above problem, a radically different technique, called `PlanBouquet`, was proposed in [4, 5]. In this approach, the highly brittle selectivity estimation process is completely *abandoned*, and replaced instead with a calibrated *discovery* mechanism. A key benefit of the new construction is that *provable guarantees* are obtained on the worst-case performance. In a follow-up work, an improved version of `PlanBouquet`, called `SpillBound` (SB), which significantly accelerates the selectivity discovery process, and provides platform-independent performance guarantees, was recently presented in [12].

`SpillBound` begins with constructing a multi-dimensional *Error-prone Selectivity Space* (ESS) at query compile-time, with each dimension corresponding to the selectivity of a specific error-prone predicate appearing in the query, and ranging over $(0, 1]$. A sample 2D ESS is shown in Figure 2 for the example query of Figure 1, with the two join predicates being treated as error-prone.

On this ESS space, a series of *isocost* contours, $\mathcal{IC}_1$ through $\mathcal{IC}_m$, are drawn – each isocost contour $\mathcal{IC}_i$ has an associated optimizer estimated cost CC_i, and represents the connected selectivity curve along which the cost of the optimal plan is CC_i. Further, the contours are selected such that the cost of the first contour $\mathcal{IC}_1$ corresponds to the minimum query cost C at the origin of the space, and the cost of each of the following contours is *double* that of the previous contour. Therefore, in Figure 2, there are five hyperbolic
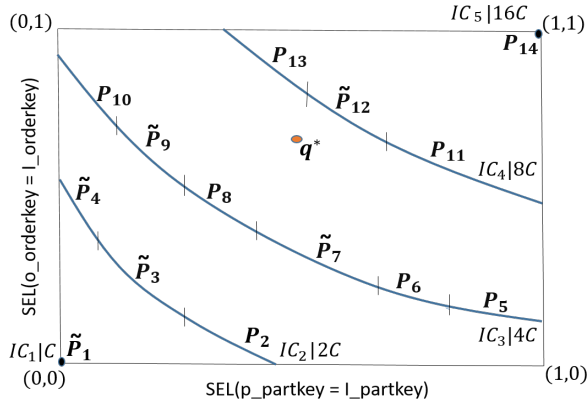
Figure 2: SpillBound Execution on 2D ESS

contours, $\mathcal{IC}_1$ through $\mathcal{IC}_5$, with their costs ranging from $\text{CC}_1 = C$ to $\text{CC}_5 = 16C$.

The union of the plans appearing on all the contours constitutes the "plan bouquet" for the query – accordingly, plans P_1 through P_{14} form the bouquet in Figure 2. Given this set, the SB run-time algorithm operates as follows: Starting with the cheapest contour $\mathcal{IC}_1$, a carefully chosen *subset* of plans on each successive contour is sequentially executed, *with the individual executions having time limits equal to the associated contour's cost*. The choice of plans is such that each execution focuses on incrementally learning the selectivity of a specific error-prone predicate. Further, the plans are executed in "spill-mode", which ensures that the assigned time budget is maximally utilized towards selectivity discovery along the chosen dimension. This process of contour-wise plan executions ends when all the selectivities in the ESS have been fully discovered. Armed with this complete knowledge, the genuine optimal plan is now correctly identified, and used to finally execute the query to completion.

To make the SB methodology concrete, consider the case where the query happens to be actually located at q^*, in the intermediate region between contours $\mathcal{IC}_3$ and $\mathcal{IC}_4$, as shown in Figure 2. Assume that the optimal plan for this location, P_{q^*}, would cost $7C$ to process the query. In contrast, SB, which is unaware of the true location, would invoke the following budgeted execution sequence:

$$P_1|C, P_3|2C, P_4|2C, P_7|4C, P_9|4C, P_{12}|8C, P_{q^*}|7C$$

In this sequence, the executions up to P_{12} are in spill-mode and determine the location of q^*, whereas the final plan P_{q^*} executes the query to completion. (For ease of visualization, the chosen subset of plans in each contour are annotated with the $\sim$ symbol in Figure 2). In this scenario, the cumulative execution cost incurred by SB is $(C + 2C + 2C + 4C + 4C + 8C + 7C) = 28C$. Since the cost of the optimal plan P_{q^*} is $7C$, the resultant sub-optimality ratio of SB for the q^* location is $28C/7C = 4$.

The surprising outcome of the above "trial-and-error" selectivity discovery strategy is that the additional execution costs can be *bounded* relative to the optimal, *irrespective of the query location in the space*. Specifically, let us use Maximum Sub-Optimality (**MSO**), as defined in [5], to capture the worst-case sub-optimality ratio of a query processing algorithm over the entire selectivity space. Then, the MSO of SpillBound is bounded by

$$MSO_{SB} \leq D^2 + 3D \qquad (1)$$

where D is the dimensionality of the ESS, i.e. the number of error-prone predicates in the input query.

Limitation of SpillBound. Notwithstanding SpillBound's unique benefits with regard to robust query processing, a major limitation is that its MSO guarantee is predicated on expending enormous pre-processing overheads during query compilation. Specifically, identifying the isocost contours in the ESS entails, in principle, $\Theta(r^D)$ calls to the query optimizer, where r is the resolution (i.e. discretization granularity) along each dimension of the ESS. So, for instance, if $r = 100$, corresponding to selectivity characterization at 1% intervals, and D is 4, a *hundred million* optimizer invocations have to be carried out to identify the contours before SB can begin executing the query. As a consequence, SB is currently suitable only for *canned queries* that are repeatedly invoked by the parent applications.

An obvious first step towards addressing the above issue is to utilize multi-core computing platforms to leverage the intrinsic parallelism available in contour identification. However, this may not be sufficient to fully address the strong exponential dependence on dimensionality. In our view, adapting the SB methodology for ad-hoc queries requires, in addition to hardware support, *algorithmic approaches* to substantively reduce the compilation overheads – the design of such approaches forms the focus of this paper.

Problem Formulation. Specifically, we investigate the trade-off between the two key attributes of the SB approach, namely, the *compilation overheads* and the *MSO guarantee*. The overhead of SB is measured as the number of optimization calls made to the query optimizer in order to construct all the isocost contours. Given an algorithmic approach aimed at reducing these compilation overheads, we use γ (≥ 1) to denote its overheads reduction factor relative to that of SB. Bringing down the overheads may, however, result in a *weaker* MSO guarantee. We use η (≥ 1) to denote this MSO relaxation factor relative to SB. With this characterization, the formal problem addressed in this paper is the following:

Given a user query Q for which SpillBound *provides an MSO guarantee M, and a user-permitted relaxation factor η on this guarantee, design a query processing algorithm that maximizes γ while ensuring that the MSO guarantee remains within ηM.*

Algorithmic Reduction of the Overheads. The MSO guarantee of SB (Equation 1) is predicated on the standard assumption that plan cost functions are *monotonically increasing* with regard to the predicate selectivities. In this paper, we leverage the stronger fact that plan cost functions typically exhibit a *concave down* behavior in the ESS – i.e. they have monotonically non-increasing *slopes*.[1] Specifically, we design a modified algorithm, FrugalSpillBound (FSB), that incorporates the concave behavior to substantially reduce the compilation overheads at the cost of a mild relaxation on the MSO guarantee. Quantitatively, the attractive tradeoff between η and γ is the following:

$$\gamma = r/\log_\eta r \qquad\qquad D = 1$$
$$\gamma = \Omega(r^D/(D\log_\eta r)^{D-1}) \qquad D \geq 2 \qquad (2)$$

That is, the initial regime of FSB provides an *exponential improvement* in γ for a linear increase in η.

More concretely, a sample instance of the $\eta - \gamma$ tradeoff is shown in the red line of Figure 3, obtained for a 4D ESS derived from Query 26 of the TPC-DS benchmark. In this figure, which graphs a *semi-log* plot, the initial exponential overhead reduction regime is long enough that a **two orders of magnitude improvement** in γ is achieved with an η of 2. Further, when *empirically* evaluated,

---

[1]As explained in Section 2, a weaker form of concavity, called Axis-Parallel Concavity, is sufficient for our techniques to hold.
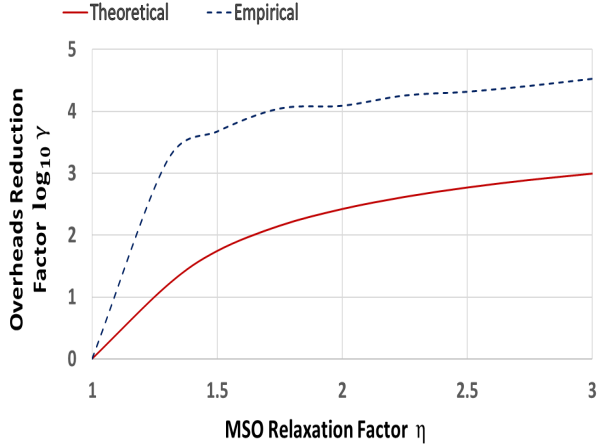
Figure 3: FSB $\eta - \gamma$ Tradeoff for 4D_Q26

the decrease in overheads is much greater – this is shown in the blue line of Figure 3, where **nearly four orders of magnitude improvement** in γ is achieved for $\eta = 2$.

The concavity assumption directly leads to an elegant FSB construction for the base case of a one-dimensional ESS. To handle the multi-dimensional scenario, however, we need additional machinery, called *bounded contour-covering sets* (BCS) – these sets serve as low-overhead replacements for the original isocost contours. More precisely, a BCS is a set of locations that collectively *spatially dominate* all locations on the associated contour, and whose costs are within a bounded factor of the contour cost. Efficient identification of the BCS is made possible thanks to the concavity assumption, and the aggregate cardinality of the BCS over the contours is *exponentially smaller* than the number of locations in the ESS, resulting in the substantially decreased overheads.

Performance Results. To demonstrate that the example $\eta - \gamma$ tradeoff for FSB shown in Figure 3 is not an isolated instance, we have carried out similar evaluations on a representative set of OLAP queries sourced from the TPC-DS benchmark, operating on the PostgreSQL engine. The query suite covers a variety of ESS dimensionalities, going up to as many as 5 dimensions, and capturing environments that are challenging from a robustness perspective. Our performance results indicate that a two orders of magnitude theoretical reduction in overheads is *routine* with $\eta = 2$, while the empirical reduction in overheads is typically an order of magnitude more than this guaranteed value, delivering a cumulative benefit of more than three orders of magnitude. Therefore, the new FSB approach represents a substantive step towards practically achieving robust query processing for ad-hoc queries with moderate ESS dimensionalities – especially in conjunction with contemporary multi-core architectures that exploit the inherent parallelism in the ESS construction. So, for instance, a 5D query which takes a **few days** even on a well-provisioned multi-core machine to complete the 10 billion optimizer calls required for constructing the entire ESS (at a resolution of 100), can now be made ready for execution within a **few minutes** by FrugalSpillBound!

Organization. The remainder of this paper is organized as follows: In Section 2, the background concepts and behavioral assumptions related to robust query processing are enumerated. The 1D version of FSB and its analysis are presented in Section 3. Subsequently, the design of FSB for 2D ESS, incorporating the BCS

machinery, is described in Section 4. This is followed by the extension to arbitrary dimensions, outlined in Section 5. The experimental framework and performance results are highlighted in Section 6. Pragmatic deployment aspects are discussed in Section 7, and the related literature is reviewed in Section 8. Finally, our conclusions are summarized in Section 9.

2. BACKGROUND

We begin by reviewing the key concepts and assumptions underlying our approach to robust query processing [5, 12].

2.1 Error-prone Selectivity Space (ESS)

Given an SQL query, any predicate whose selectivity is difficult to estimate accurately is referred to as an *error-prone predicate*, or epp. For a query with D epps, the set of all epps is denoted by EPP $= \{e_1, \ldots, e_D\}$, where e_j denotes the jth epp. The selectivities of the D epps are mapped to a D-dimensional space, with the selectivity of e_j corresponding to the jth dimension. Since the selectivity of each predicate ranges over $(0, 1]$, a D-dimensional hypercube $(0, 1]^D$ results, henceforth referred to as the *error-prone selectivity space*, or ESS. Note that each location $q \in (0, 1]^D$ in the ESS represents a specific query instance where the epps happen to have the selectivities corresponding to q. Accordingly, the selectivity value of q on the jth dimension is denoted by $q.j$.

For tractability, the ESS is discretized at a fine-grained resolution r in each dimension. We refer to the location corresponding to the minimum selectivity in each dimension as the *origin* of the ESS, and the location at which the selectivity value in each dimension is maximum as the *terminus*. In our framework, the origin and the terminus correspond to query locations with $q.j = 1/r \ \forall j$ and $q.j = 1 \ \forall j$, respectively.

2.2 POSP Plans

The optimal plan for a generic selectivity location $q \in$ ESS is denoted by P_q, and the set of such optimal plans over the complete ESS constitutes the *Parametric Optimal Set of Plans* (POSP) [9].[2] We denote the cost of executing any plan P at a selectivity location $q \in$ ESS by $Cost(P, q)$. Thus, $Cost(P_q, q)$ represents the optimal execution cost for the selectivity instance located at q. Further, we assume that the query optimizer can identify the optimal query execution plan if the selectivities of all the epps are correctly known.[3] For ease of presentation, we will hereafter use *cost of a location* to refer to the cost of the optimal plan at that location.

2.3 Optimal Cost Surface (OCS)

The trajectory of the minimum cost plan through the entire D-dimensional ESS represents the Optimal Cost Surface (OCS) – an example for a 2D ESS is shown in Figure 4, where the X and Y axes represent the two join predicate selectivities, and the Z axis represents the cost of each location in this selectivity space. The intersection of the isocost hyperplanes ($\mathcal{IC}_1$ through $\mathcal{IC}_5$) with the OCS, which results in the isocost contours, is also captured in Figure 4. In fact, the projected isocost contours shown in Figure 2 were constructed from a similar OCS.

2.4 Maximum Sub-Optimality (MSO)

We now move on to describing the MSO performance metric proposed in [5] to quantify the robustness of query processing. For

---

[2]Letter subscripts for plans denote locations, whereas numeric subscripts denote identifiers.

[3]For example, through the classical Dynamic Programming-based search of the plan enumeration space [20].
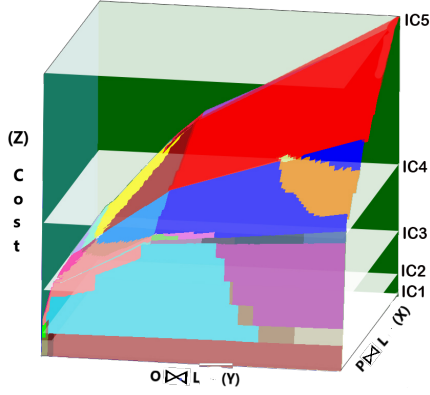
Figure 4: Optimal Cost Surface (OCS)

this purpose, let q_a denote the ESS location corresponding to the actual selectivities of the user query epps – note that this location is unknown at compile-time, and needs to be explicitly discovered. As discussed in the Introduction, SpillBound carries out a sequence of budgeted plan executions in order to discover the location of q_a. We denote this sequence by Seq_{q_a}, with each element s_i in the sequence being a pair, (P_i, ω_i) indicating that plan P_i is executed with a maximum time budget of ω_i.

The sub-optimality of this plan sequence is defined relative to an oracle that magically knows the correct query location apriori and therefore directly uses the ideal plan P_{q_a}. That is,

$$SubOpt(\text{Seq}_{q_a}) = \frac{\sum\limits_{s_i \in \text{Seq}_{q_a}} \omega_i}{Cost(P_{q_a}, q_a)}$$

from which we derive

$$MSO = \max_{q_a \in \text{ESS}} SubOpt(\text{Seq}_{q_a})$$

In essence, MSO represents the *worst-case* suboptimality that can occur with regard to plan performance over the entire ESS space.

2.5 Plan Cost Monotonicity (PCM)

The notion of a location q_1 *spatially dominating* a location q_2 in the ESS plays a central role in our robust query processing framework. Formally, given two distinct locations $q_1, q_2 \in \text{ESS}$, q_1 spatially dominates q_2, denoted by $q_1 \succ q_2$, if $q_1.j \geq q_2.j$ for all $j \in \{1, \ldots, D\}$. Given spatial domination, an essential assumption that allows SpillBound to systematically explore the ESS is that the cost functions of the plans appearing in the ESS all obey *Plan Cost Monotonicity* (PCM). This constraint on plan cost function (PCF) behavior may be stated as follows: For any pair of distinct locations $q_b, q_c \in \text{ESS}$, and for any plan P,

$$q_b \succ q_c \Rightarrow Cost(P, q_b) > Cost(P, q_c)$$

That is, it encodes the intuitive notion that when more data is processed by a query, signified by the larger selectivities for the predicates, the cost of the query processing also increases. In a nutshell, *spatial domination implies cost domination*.

2.6 Axis-Parallel Concavity (APC)

We augment the above PCM assumption with a stricter condition in this paper, wherein the PCFs are not only monotonic, but also exhibit a weak form of *concavity* in their cost trajectories.

In the 1D world, a plan cost function $\mathcal{F}_p$ is said to be concave if, for any pair of locations q_1, q_2 in the 1D ESS, and any $\alpha \in [0, 1]$,

$$\mathcal{F}_p((1 - \alpha)q_1 + \alpha q_2) \geq (1 - \alpha)\mathcal{F}_p(q_1) + \alpha\mathcal{F}_p(q_2) \quad (3)$$

Generalizing to D dimensions, a PCF $\mathcal{F}_p$ is said to be *axis-parallel concave* (APC) if the function is concave along every axis-parallel 1D segment of the ESS. That is, Equation 3 is satisfied by any generic pair of locations q_1, q_2 in the ESS that belong to a common 1D segment of the ESS (i.e., $\exists j$ s.t. $q_1.k = q_2.k, \forall k \neq j$). So, for example, if e_1 and e_2 are the epps of a 2D ESS, then the APC requirement is that each PCF should be concave along every vertical and horizontal line in the ESS.

Note that APC is a strictly *weaker* condition than complete concavity across all dimensions – that is, all fully-concave functions satisfy APC, but the reverse may not be true. Further, an important and easily provable implication of the PCFs exhibiting APC is that the corresponding OCS, which is the infimum of the PCFs, *also satisfies* APC. Finally, for ease of presentation, we will generically use concavity to mean APC in the remainder of this paper.

Empirical Validation of APC

An immediate question that arises in the above context is whether the concavity assumptions on the PCFs (and, by implication, the OCS) generally hold true in practice. For this purpose, we have carried out extensive experimental evaluation with the TPC decision-support benchmarks operating on contemporary database engines. The summary finding of this empirical evaluation, whose details are presented later in Section 6, is that APC is consistently observed over almost the entire ESS.

As a sample instance, the axis-parallel projections of the 2D OCS presented in Figure 4, are computed in Figures 5a and 5b for the $Part \bowtie Lineitem$ and $Orders \bowtie Lineitem$ join predicates, respectively. These figures are graphed on a *log-log scale* and for ease of representation, capture only the optimality region of each PCF. We observe here that the PCFs clearly exhibit concavity in their optimality regions with respect to selectivity. As a direct consequence, the OCS exhibits concavity over the entire selectivity range, justifying the assumption on which our results are based.
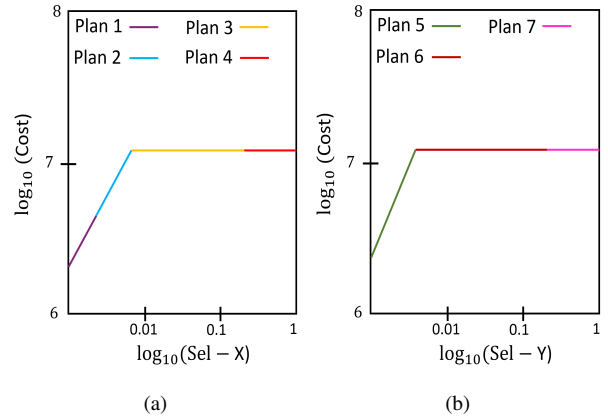

(a)

(b)

Figure 5: Validation of Axis-Parallel Concavity

A detailed rationale as to why PCF and OCS concavity is typically expected for contemporary disk-resident enterprise warehouses is given in [11]. However, we hasten to note that there are a few *operator* cost functions (e.g. Sort, Caching [15], Branching [13]) that may violate this assumption, especially in the context

of main-memory database systems. Notwithstanding these localized operator violations, it is still possible for the global OCS to remain essentially concave since the contributions of these operators to the overall plan costs may be relatively minor – for instance, the contribution of Sort to plan costs is found to be miniscule in [6].

2.7 Compilation Overheads

As mentioned in the Introduction, we measure the query compilation overheads in terms of the number of optimization calls made to the underlying database engine. With regard to this metric, the overheads incurred by SB in constructing the ESS can be computed as follows: SB first computes the optimal plans for *all* locations in the discretized ESS grid. This is carried out through repeated invocations of the optimizer with different selectivity values and combinations. Then, the isocost contours are drawn as connected curves on this discretized diagram. So, if we assume a grid resolution of r in each dimension of the ESS, the total number of optimization calls required by this approach is r^D.

Note, however, that we do not require the complete characterization of the ESS, but only the portions related to the isocost contours, as shown in Figure 2. An optimized variant, called **Nexus**, was proposed in [5] to implement this observation, and shown to provide material reductions in the contour identification overheads. However, from a deployment perspective, there are several challenges in practically using Nexus, as detailed in [11]. We have therefore chosen to instead simply assume that the entire ESS is enumerated, and consequently r^D is used to represent the baseline SB overheads in the sequel. Further, note that FrugalSpillBound is *not* impacted by such deployment issues since its compilation efforts are carried out afresh at each ad-hoc query's submission time.

2.8 Notations

For easy reference, the notations used in the remainder of the paper are summarized in Table 1.

Table 1: NOTATIONS

Notation	Meaning
epp (EPP)	Error-prone predicate (its collection)
ESS	Error-prone selectivity space
D	Number of dimensions in the ESS
r	Grid resolution in each ESS dimension
$e_1, \ldots, e_D$	The D epps in the query
$q \in [0,1]^D$	A query location in the ESS
$q.j$	Selectivity of q in jth ESS dimension
P_q	Optimal Plan at q
q_a	Actual query location in ESS
$Cost(P, q)$	Cost of plan P at location q
$\mathcal{IC}_i$	Isocost Contour i
CC_i	Cost of an isocost contour $\mathcal{IC}_i$
BCS_i	Bounded contour-covering set of contour $\mathcal{IC}_i$
η	User-specified MSO relaxation factor
γ	Reduction factor wrt compilation overheads

3. FRUGAL SPILLBOUND FOR 1D ESS

1D SB. We begin by reviewing how the SpillBound algorithm operates on a 1D ESS. The sample concave OCS function $\mathcal{F}$, shown in Figure 6, is used to aid the description. In this figure, the selectivity axis represents the selectivity range for the lone

epp, and the cost axis represents the OCS function. The cost axis is discretized into doubling-based isocost contours, $\mathcal{IC}_1$ through $\mathcal{IC}_m$, with $CC_i = 2^{i-1}C$. Note that, in the case of 1D OCS, each of the contours correspond to a *single* selectivity location on the selectivity axis. We denote the location corresponding to $\mathcal{IC}_i$ by Q_i. Further, $Q_1 = 1/r$ and $Q_m = 1$ correspond to the origin and terminus locations, respectively.
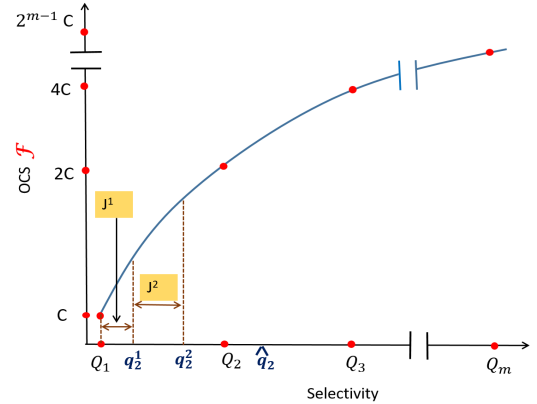

Figure 6: Concave OCS

Conceptually, the 1D-SB algorithm has two phases, a *compilation phase* and an *execution phase*. During the compilation phase, for each of the r uniformly spaced locations on the selectivity axis, the optimal plan at the location and its cost are determined. Using the cost information from the r locations, the precise location of Q_i is identified for $i = 1, \ldots, m$. The set of optimal plans at the Q_i locations is called the "bouquet of plans". Then, during the execution phase, this bouquet of plans is sequentially executed, starting from the cheapest isocost contour, with a budget equal to the associated contour cost. The process ends when a plan reaches completion within its allocated budget. As proved in [5], this budgeted sequence of plan executions achieves an MSO guarantee of 4 with a compilation overhead of r optimizer calls.

We now move on to presenting our 1D-FSB algorithm, which also has compilation and execution phases, as described below.

3.1 Compilation Phase

The main idea in the compilation phase of FSB is to dispense with SB's approach of precisely identifying the location of the Q_is. Instead, for each Q_i, we identify a *proxy location*, $\hat{q}_i$, such that the cost of the optimal plan at $\hat{q}_i$ is in the range $[\mathcal{F}(Q_i), \eta\mathcal{F}(Q_i)]$. The compilation phase consists of identifying these proxy locations $\hat{q}_i$s via a sequence of calibrated *jumps* in the selectivity space, as described next.

Discovering the proxy for Q_2

Since Q_1 is known, we set $\hat{q}_1 = Q_1$. The search for $\hat{q}_2$ starts from $\hat{q}_1$. We now perform a sequence of jumps in the selectivity space until we land exactly at Q_2 or overshoot it for the first time. Further, the lengths of the jumps are calibrated such that when $\hat{q}_2$ is reached, its cost is guaranteed to be in the range $[\mathcal{F}(Q_2), \eta\mathcal{F}(Q_2)]$, as described below.

First Jump. Identify the optimal plan $P_{\hat{q}_1}$ at $\hat{q}_1$, and compute its slope, $s(\hat{q}_1)$ at $\hat{q}_1$.[4] The slope is calculated through plan recosting[5] of $P_{\hat{q}_1}$ at a selectivity location in the close neighborhood of $\hat{q}_1$.

Our first estimate for $\hat{q}_2$, denoted by q_2^1 (refer to Figure 6), is the location that is expected to have η times the cost of $\mathcal{F}(\hat{q}_1)$ when extrapolated by a tangent line with a slope of $s(\hat{q}_1)$, i.e,

$$\frac{\mathcal{F}(\hat{q}_1) + s(\hat{q}_1) \cdot (q_2^1 - \hat{q}_1)}{\mathcal{F}(\hat{q}_1)} = \eta$$

By rearranging, we get

$$q_2^1 = \hat{q}_1 + \frac{(\eta - 1) \cdot \mathcal{F}(\hat{q}_1)}{s(\hat{q}_1)}$$
$$\therefore q_2^1 = \hat{q}_1 + J^1$$

where J^1 represents the first jump towards $\hat{q}_2$, relative to the starting location, $\hat{q}_1$. The following lemma immediately follows from the concavity of the PCF:

LEMMA 3.1. *The cost condition $\mathcal{F}(q_2^1) \leq \eta \cdot \mathcal{F}(\hat{q}_1)$ is satisfied.*

We next show that the jump J^1 is such that the selectivity of q_2^1 is at least η times the selectivity at $\hat{q}_1$.

LEMMA 3.2. *The selectivity of q_2^1 is at least η times the selectivity at $\hat{q}_1$, i.e, $\mathbf{q_2^1 \geq \eta \hat{q}_1}$.*

PROOF. Let the tangent of $\mathcal{F}$ at $\hat{q}_1$ be expressed as

$$\mathcal{F}(q) = s(\hat{q}_1) \cdot q + c' \qquad 0 \leq q \leq 1, c' \geq 0$$

(Here, $c' \geq 0$ to ensure non-negative cost at $q = 0$.) Based on this equation, we obtain the following pair of equations by separately considering the PCF cost at $\hat{q}_1$ and the estimated cost at q_2^1 .

$$\mathcal{F}(\hat{q}_1) = s(\hat{q}_1) \cdot \hat{q}_1 + c'$$
$$\eta \cdot \mathcal{F}(\hat{q}_1) = s(\hat{q}_1) \cdot q_2^1 + c'$$

Simplifying the equation pair, we get

$$\eta s(\hat{q}_1) \cdot \hat{q}_1 + \eta c' = s(\hat{q}_1) \cdot q_2^1 + c'$$
$$\therefore q_2^1 = \eta \hat{q}_1 + \frac{(\eta - 1)c'}{s(\hat{q}_1)} \geq \boldsymbol{\eta \hat{q}_1}$$

□

Depending on the cost at the first jump's landing location, i.e. at q_2^1, two cases are possible:

1. *Cost Overshoot*, i.e, $\mathcal{F}(q_2^1) \geq \mathcal{F}(Q_2)$: In this case, we have identified a proxy location for Q_2 whose cost is at most $\eta \mathcal{F}(Q_2)$ (by Lemma 3.1).

2. *Cost Undershoot*, i.e., $\mathcal{F}(q_2^1) < \mathcal{F}(Q_2)$: In this case, the jump scheme is repeated with q_2^1 as the starting location. That is, we jump to q_2^2, with the jump length being $J^2 = \frac{(\eta - 1)\mathcal{F}(q_2^1)}{s(q_2^1)}$. This process is repeated until we reach $\hat{q}_2$, signalled by $\mathcal{F}(\hat{q}_2) \geq \mathcal{F}(Q_2)$. Since the cost of $\hat{q}_2$'s previous location is less than $\mathcal{F}(Q_2)$, Lemma 3.1 guarantees that $\mathcal{F}(\hat{q}_2) \leq \eta \mathcal{F}(Q_2)$.

---

[4]Due to PCM, the slope at any location in $\mathcal{F}$ is > 0.

[5]Recosting is an engine feature that costs an abstract plan for a query, and is around 100 times faster than optimizer calls [6].

3.1.1 *Implementation of Proxy Discovery*

The above compilation phase of FSB for the 1D scenario is detailed in Algorithm 1. Here, the entire search from $\hat{q}_1$ to $\hat{q}_2$ is captured as a generic Explore subroutine, with three arguments: seed, the starting location, t_cost, the cost at the terminal location, and r_factor, the relaxation factor wrt t_cost.

The proxy location $\hat{q}_i$ for Q_i is obtained starting with the proxy location $\hat{q}_{i-1}$. This is done by calling the Explore subroutine, with seed as $\hat{q}_{i-1}$, target cost of CC_i, and relaxation factor of η. The derivation that bounded the relative cost of $\hat{q}_2$ w.r.t. the cost of Q_2 can be repeated to show that the cost of $\hat{q}_i$ is at most $\eta \mathcal{F}(Q_i)$ for $i = 2, \ldots, m - 1$. Finally, the output of the algorithm is a set of proxy locations, ProxyContourLocs = $\{Q_1 \cup \{\bigcup_{i=2}^{m-1} \hat{q}_i\} \cup Q_m\}$.

3.1.2 *Bounded Compilation Overheads*

THEOREM 3.3. *The compilation overheads reduction, γ, of* 1D-FSB *is at least* $\frac{r}{\log_\eta r}$ *.*

PROOF. From Lemma 3.2, the maximum number of jumps is required when the selectivity estimation at each jump is exactly η times the selectivity of the previous location. Therefore, the total number of query optimizer calls is bounded as follows:

$$\text{Total Optimization Calls} \leq \log_\eta \frac{Q_m}{Q_1} \leq \log_\eta r$$

Thus, the compilation overheads reduce from r to $\log_\eta r$.   □

3.2 Execution Phase

The execution phase of FSB, as shown in Algorithm 1, is the same as that of SB except that the plan bouquet now consists of the optimal plans at the proxy locations in ProxyContourLocs. We therefore easily derive the following theorem for *maintaining the η constraint*.

THEOREM 3.4. *The MSO relaxation of* 1D-FSB *is at most η.*

PROOF. From the compilation phase, we know that the cost of a proxy location $\hat{q}_i$ is at most η times the cost of Q_i. The bounded cost of each proxy location ensures that the sequence of execution costs for the 1D-FSB plan bouquet is $C, 2\eta C, 4\eta C, \ldots$ (as opposed to $C, 2C, 4C, \ldots$ for 1D-SB). Since the MSO of 1D-SB is 4, it follows that the MSO of 1D-FSB is bounded by 4η.   □

4. FRUGAL SPILLBOUND FOR 2D ESS

In this section, we present the extension of 1D-FSB to the 2D case. For ease of exposition, we refer to the two epps as x and y, respectively.

In the 1D ESS, each contour was a single point. However, in 2D, it is a continuous *curve* as shown in Figure 7. Therefore, the step of identifying the proxy locations for Q_is has to be generalized so as to *cover* an isocost contour $\mathcal{IC}_i$ with an appropriate set of proxy locations. We achieve this by finding a *bounded contour-covering set* (BCS) of locations for each contour $\mathcal{IC}_i$. The definition of these sets and their identification procedure are presented next.

4.1 Bounded Contour-covering Set (BCS)

The BCS for a contour is defined as the set of locations such that:

(a) Every location in the contour is *spatially dominated* by at least one location in this set; and

(b) The cost of each location in BCS is *bounded* to within an η factor of the contour cost.

Algorithm 1 `1D-FSB` (η)

1: **Compilation Phase:**
2: set $Q_1 = 1/r$ and $Q_m = 1$;
3: set $k = 2$;
4: set ProxyContourLocs = $\{Q_1, Q_m\}$;
5: set $\hat{q}_1 = Q_1$;
6: **while** $k < m - 1$ **do**
7:     $\hat{q}_k$ = Explore($\hat{q}_{k-1}$, CC$_k$, η);
8:     Add $\hat{q}_k$ to ProxyContourLocs;
9:     k++;
10: **end while**

11: **function** Explore(seed, t_cost, r_factor);
12: compute $cost = \mathcal{F}(\text{seed})$ (using optimizer call);
13: **while** $cost < $ t_cost **do**
14:     compute $slope$ at seed (using plan recosting);
15:     $next\_jump = (\text{r\_factor} - 1) \cdot \dfrac{cost}{slope}$;
16:     seed $+ = next\_jump$;
17:     $cost = \mathcal{F}(\text{seed})$;
18: **end while**
19: **return** seed;
20: **end function**

21: **Execution Phase:**
22: **for** q in ProxyContourLocs **do**
23:     Execute optimal plan P_q with budget $\mathcal{F}(q)$;
24:     **if** P_q completes execution **then**
25:       Return query result;
26:     **else**
27:       Terminate P_q and discard partial results;
28:     **end if**
29: **end for**

We denote the BCS of contour $\mathcal{IC}_i$ by BCS_i. Formally, BCS_i is a set that needs to satisfy the following condition:

$\forall q \in \mathcal{IC}_i, \exists\, q' \in BCS_i$ such that
$$q \preceq q' \text{ and } Cost(P_{q'}, q') \leq \eta \text{CC}_i$$

To make this notion concrete, a candidate BCS_i for the example contour $\mathcal{IC}_i$ shown in Figure 7, is $\{c_1, c_2, c_3\}$ which covers the entire contiguous length of the contour. As a specific case in point, the covering location c_2 fully covers the optimality segments of P_5 and P_6, as well as parts of P_4 and P_7, in $\mathcal{IC}_i$.
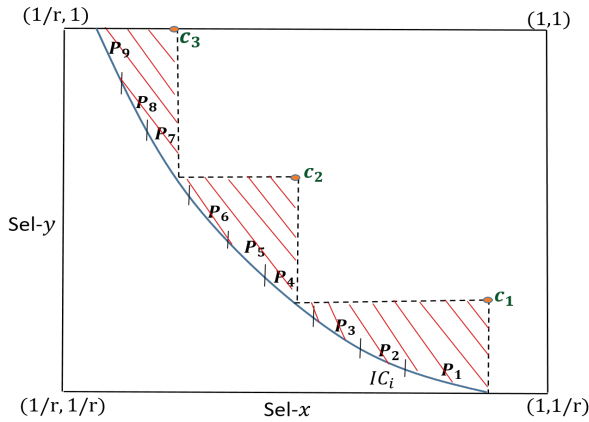

Figure 7: Bounded Contour-covering Set (BCS)

4.2 Compilation Phase

We now present a computationally efficient method to find a BCS for an isocost contour in the 2D `ESS`. To generalize the 1D method, we carry out jumps in the selectivity space along *both* the x and y dimensions. These jumps are designed to be axis-parallel and we leverage APC in their analysis. A special feature, however, is that the jumps are in *opposite* directions in the two dimensions – *forward* in one, and *reverse* (i.e. jumps are performed in the decreasing selectivity direction) in the other. Further, in the reverse jumps, the selectivity of the next location is decreased by a *constant* factor, as explained below – this is in marked contrast to the forward jumps, where the Explore (seed,t_cost,r_factor) subroutine is invoked to decide the next location.

In principle, the choice of dimensions for forward and reverse jumps can be made arbitrarily. However, for ease of presentation, we assume hereafter that all forward jumps are in the y dimension, and all reverse jumps are in the x dimension.

4.2.1 Algorithm Description

We explain the compilation phase by describing the process of constructing BCS_i for the isocost contour $\mathcal{IC}_i$ shown in Figure 7. For ease of presentation, we refer to Figure 8, which overlays the construction of BCS_i on top of contour $\mathcal{IC}_i$.

The main idea is to carry out a sequence of interleaved search steps that alternatively explore the x and y dimensions. Specifically, we start from the location $c_0 = (1, 1/r)$ as the seed, and search for a location, u_1, on $y = 1/r$ line whose cost is in the range $[\text{CC}_i, \sqrt{\eta}\text{CC}_i]$. A sequence of reverse jumps from c_0, with constant $\sqrt{\eta}$ factor decrease in selectivity each time, is carried out until we reach u_1. The Explore subroutine is now invoked along the increasing y dimension with u_1 as the seed location, terminating cost $\sqrt{\eta}\text{CC}_i$, and relaxation factor $\sqrt{\eta}$. Let the location returned be c_1, and by the construction of Explore, we know that its cost is in the range $[\sqrt{\eta}\text{CC}_i, \eta\text{CC}_i]$. Now, starting from c_1, a sequence of reverse jumps, again with $\sqrt{\eta}$ selectivity decrease in each jump, is carried out till we reach a location u_2 whose cost is in the range $[\text{CC}_i, \sqrt{\eta}\text{CC}_i]$. This is followed by a call to Explore with u_2 as the seed and the same settings as before for the other arguments. The returned location is now c_2. This interleaved process of reverse jumps along the x dimension and forward jumps along the y dimension, is repeated until the process hits the boundary of the `ESS`. Let us say that the process ends at location c_k ($k = 4$ for the example contour in Figure 8). Then, the set $BCS_i = \{c_1, \ldots, c_k\}$ is returned as the BCS of contour $\mathcal{IC}_i$. This description of the compilation phase of `2D-FSB` is codified in Algorithm 2.
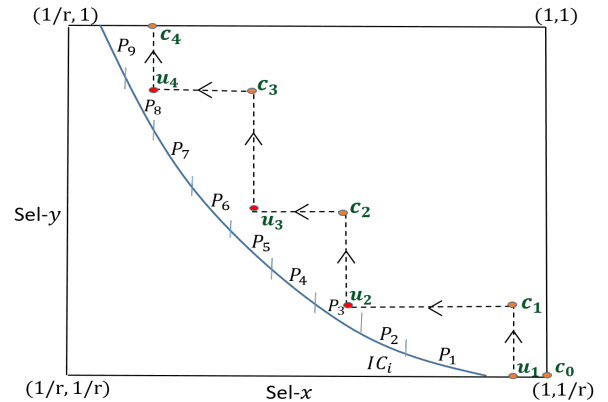

Figure 8: Identification of BCS

Algorithm 2 `2D-FSB` Algorithm (η)

1: **Compilation Phase:**
2: set $\beta = \sqrt{\eta}$;
3: **while** contours are remaining **do**
4:     set $q_{cur} = (1/r, 1/r)$;
5:     /*Let $\mathcal{IC}_i$ denote the current contour and `CC`$_i$ be its cost*/
6:     **while** $q_{cur}.x \geq \frac{1}{r}$ and $q_{cur}.y \leq 1$ **do**
7:         Find u_i with cost in $[\texttt{CC}_\texttt{i}, \beta \cdot \texttt{CC}_\texttt{i}]$ through x-axis reverse jumps;
8:         $q_{cur}.x = u_i.x$;
9:         Call Explore($u_i, \beta \cdot \texttt{CC}_\texttt{i}, \beta$) along y-axis to find c_i;
10:         $q_{cur}.y = c_i.y$;
11:     **end while**
12:     Union of all c_is forms the bounded contour-covering set, BCS_i;
13:     /* Move to next contour */
14: **end while**

15: **Execution Phase:**
16: Run the 2D `SpillBound` algorithm on the plans corresponding to BCS_i for each contour $\mathcal{IC}_i$;

4.2.2 Proof of Correctness

In order to demonstrate that every location in the contour is spatially dominated by at least one location in the associated BCS, we need to first prove that reverse jumps allow us to find u_is, whose costs are in the range $[\texttt{CC}_\texttt{i}, \sqrt{\eta}\texttt{CC}_\texttt{i}]$. Equivalently, it is sufficient to show that each reverse jump results in a relative cost decrease of at most $\sqrt{\eta}$. To do so, let us fix a covering location c_k, and let $\mathcal{F}_{ap}$ denote the restriction of OCS to the horizontal line passing through c_k. Then, we have the following lemma:

LEMMA 4.1. *The reverse jump from a location q along the x direction by a factor $\sqrt{\eta}$ results in a relative cost decrease of at most $\sqrt{\eta}$, i.e., $\mathcal{F}_{ap}(\frac{q.x}{\sqrt{\eta}}, q.y) \geq \mathcal{F}_{ap}(q.x, q.y)/\sqrt{\eta}$.*

PROOF. Let q' denote the location $(q.x/\sqrt{\eta}, q.y)$. Consider the line passing through q', parallel to the x-axis, and tangent to OCS. Let s be its slope and c' its intercept on the cost axis ($c' \geq 0$ to ensure non-negative cost at the origin). We know that this line *overestimates* the cost at q because $\mathcal{F}_{ap}$ is both increasing and concave (by virtue of its PCM and APC characteristics). Thus, we have

$$
\begin{aligned}
\mathcal{F}_{ap}(q) &\leq s \cdot (q.x) + c' \\
&\leq \sqrt{\eta}\left(s \cdot (\frac{q.x}{\sqrt{\eta}}) + \frac{c'}{\sqrt{\eta}}\right) \\
&\leq \sqrt{\eta}\left(s \cdot (\frac{q.x}{\sqrt{\eta}}) + c'\right) \\
&= \sqrt{\eta}\mathcal{F}_{ap}(q')
\end{aligned}
$$

where the second and third inequalities are implied by $\eta \geq 1$ and $c' \geq 0$. The last equality follows from the fact that the line passes through q'.   □

LEMMA 4.2. *Every location in $\mathcal{IC}_i$ is dominated by at least one location in BCS_i .*

PROOF. Consider any point q in $\mathcal{IC}_i$. By construction we know that there exists $c_k \in BCS_i$ s.t. $c_k.y \leq q.y \leq c_{k+1}.y$. We will show that $c_{k+1} \in BCS_i$ is a dominating location for q by proving $q.x \leq c_{k+1}.x$. Consider the location u_{k+1} whose x coordinate is the same as that of c_{k+1}. This means that (a) $u_{k+1}.x = c_{k+1}.x$, and (b) $u_{k+1}.y = c_k.y$. Since the cost of location u_{k+1} is $\geq$ the cost of location q, and $u_{k+1}.y \leq q.y$ by PCM, it implies that $u_{k+1}.x \geq q.x$. Therefore, q is dominated by c_{k+1}.   □

4.2.3 Bounded Computational Overheads

Now that we have shown the coverage properties of the BCS, we move on to proving that their identification can be accomplished with bounded overheads.

LEMMA 4.3. *The overheads reduction, γ, of `2D-FSB` is at least $\dfrac{r^2}{4 \cdot m \cdot \log_\eta r}$.*

PROOF. Let us first consider the number of optimization calls required per contour for `2D-FSB`. We know that the exploration of c_ks and u_ks move unidirectionally along the y-axis ($1/r$ to 1) and x-axis (1 to $1/r$), respectively. Furthermore, we have earlier shown that each jump results in a relative increase (or decrease) in selectivity of at least $\sqrt{\eta}$. Thus, by geometric progression, we can infer the following:

$$
\begin{aligned}
\text{Opt. calls per Contour} &= \text{Opt. calls for } c_k\text{s} + \text{Opt. calls for } u_k\text{s} \\
&\leq \log_{\sqrt{\eta}} r + \log_{\sqrt{\eta}} r \\
&= 2\log_{\sqrt{\eta}} r = 4 \cdot \log_\eta r
\end{aligned}
$$

Since there are m contours in the ESS, we conclude that there are $4 \cdot m \cdot \log_\eta r$ optimization calls across all contours for `2D-FSB`, as compared to r^2 for `2D-SB`. Thus, γ is at least $\dfrac{r^2}{4 \cdot m \cdot \log_\eta r}$.   □

4.3 Execution Phase

In the execution-phase, we run the original `2D-SB` algorithm, treating the BCS identified for every contour as the *effective contour*. Specifically, starting from the least cost BCS_1, the plans corresponding to the locations in each successive BCS_i are executed as per the `2D-SB` algorithm. This BCS-based execution of plans (in spill-mode) is continued until the actual selectivities of both the epps are learned. Finally, the optimal plan at the discovered selectivity location is executed to completion to compute the query results for the user. We show below that with this execution strategy, the MSO guarantee is relaxed by at most η.

4.3.1 Maintaining the η constraint

THEOREM 4.4. *The MSO relaxation of `2D-FSB` is at most η.*

PROOF. We know that the cost of any location in BCS_i is at most $\eta\texttt{CC}_\texttt{i}$. Furthermore, the execution-phase runs the 2D SB algorithm on the BCS of every contour. Thus, every execution in `2D-FSB` is performed with a budget of η times its corresponding contour cost. Hence, the overall cost of `2D-FSB` is at most η times that of 2D SB, which increases the MSO by at most η.   □

The above proof is predicated on assuming that the constituent features of the `2D-SB` algorithm are not impacted by the new constructions. In particular, the analysis of `2D-SB` in [12] relied on two crucial properties: *Half-Space Pruning* (HSP) and *Contour Density Independent Execution* (CDIE). With HSP, a single spill-mode execution of a plan is sufficient to divide the ESS into two disjoint half-spaces, and obtain evidence that q_a lies in one of them (i.e. the other half-space gets pruned). On the other hand, the CDIE property implies that at most two plan executions are required per contour, irrespective of the actual number of plans on the contour. We formally prove in [11] that both these HSP and CDI Execution properties *continue* to hold for `2D-FSB`, thereby ensuring the validity of Theorem 4.4.

5. MULTI-DIMENSIONAL FSB

In this section, we show how to extend `2D-FSB` to higher dimensions. At first glance, the potential epps in a canonical OLAP query can be large in number – for instance, we have observed as many as 12 for some TPC-DS queries. If all these epps were to be made part of the ESS, it would result in an impractically large search space that cannot be explored efficiently. Therefore, before describing the `MultiD-FSB` algorithm, we first explain how it is usually feasible, through a pre-processing step, to construct a *low-dimensional* ESS from the initial large candidate set – the important point to note is that our dimensionality reduction is achieved *without* impacting the MSO guarantees of the query. Due to space constraints, we only summarize the approach here – the complete details are available in [19].

5.1 Constructing Low-Dimensional ESS

The key idea in our preprocessing step, called **DimRed**, is to associate an *impact factor* with each candidate epp. The impact factor of an epp e is defined as the worst-case relative inflation in the cost of POSP plans when the selectivity of e is varied over the entire range from $1/r$ to 1, while keeping the selectivities of other predicates fixed. Now consider a predicate e with impact factor f – if we drop e from the EPP, the reduced ESS can be sub-optimal by at most a factor $(1 + f)$ w.r.t. the original ESS that contained e; at the same time, the MSO also decreases due to the reduced ESS dimensionality. Therefore, we consider the epps in the increasing order of their impact factors, and incrementally keep dropping them while the net benefit of the reduced dimensionality on the MSO guarantee is more than the net sub-optimality incurred by the dropped predicates. The final set of retained predicates constitutes the relevant ESS for the `MultiD-FSB`.

A detailed description of how the entire DimRed reduction algorithm, including the identification of epp impact factors, can be implemented efficiently is provided in [19]. For instance, the dimensionality of TPC-DS Q91 was reduced by DimRed from 12 to 5 in less than 20 seconds.

5.2 Multi-D Algorithm

The `MultiD-FSB` algorithm is executed on the set of dimensions retained after the DimRed pre-processing step. The retained epps are first ordered in decreasing value of their impact factors, i.e. e_1 has the highest impact factor and e_D, the lowest. Then, for every contour, we construct a specially designed sparse grid $\mathcal{G}$ for the first $(D - 2)$ dimensions. $\mathcal{G}$ is constructed such that there are totally $(\log_\beta(1/r))^{D-2}$ points, where $\beta = \sqrt[D]{\eta}$. Further, in each dimension, there are $\log_\beta(1/r)$ points that are spread out in a geometric distribution with factor β. The key feature of this grid is that, even if we restrict the search space in the first $D - 2$ dimensions to just the points in $\mathcal{G}$, we incur an MSO relaxation factor no more than β^{D-2} while still ensuring complete coverage of the underlying contour – the proof of this claim is available in [11].

The algorithm to cover an isocost contour $\mathcal{IC}_i$ runs in two important steps:

S1: Corresponding to each point p in the grid $\mathcal{G}$, run a `2D-FSB` with the first $(D - 2)$ dimensions fixed as per p, and the last two dimensions at the full resolution of r. The output is treated as a subset of BCS_i.

S2: Compute the union of the `2D-FSB` outputs obtained in step **S1** over all the points in $\mathcal{G}$ – this union forms the final BCS_i.

Each invocation of `2D-FSB` incurs an MSO relaxation factor of β^2 corresponding to the last two dimensions. Further, $\mathcal{G}$ contributes

an MSO relaxation factor of β^{D-2} due to the first $(D - 2)$ dimensions. Thus, the overall MSO relaxation is contained at β^D, i.e. η. The pseudocode for `MultiD-FSB` is presented in Algorithm 3, and its analysis leads to the following theorems on overheads reduction and on maintenance of the η relaxation constraint – their proofs are available in [11].

THEOREM 5.1. *The compile-time overheads reduction, γ, of* `MultiD-FSB` *is at least* $r^D/(2 \cdot m \cdot (D \cdot \log_\eta r)^{D-1})$.

THEOREM 5.2. *The MSO relaxation of* `MultiD-FSB` *is at most η.*

Algorithm 3 `MultiD-FSB` (η)

1: **Compilation Phase:**
2: Set: $\beta = \sqrt[D]{\eta}$;
3: Set: $k = 1$; /*initialization to first contour*/
4: **while** contours are remaining **do**
5:     Set: $q_{cur} = (\frac{1}{r}, \cdots, \frac{1}{r})$; /*starting to explore the kth contour*/
6:     $BCS_k = \emptyset$
7:     **for** $q_{cur}.1 = \frac{1}{r}$; $q_{cur}.1 \leq 1$; $q_{cur}.1 = \beta q_{cur}.1$ **do**
8:         /* $D - 3$ *more nested* **for** *loops like the above corresponding to the dimensions* 2 *through* $(D - 2)$*/
9:         /* At the end of $(D - 2)$ nested **for** loops, q_{cur} is such that its first $(D - 2)$ dimensions correspond to one of the points in the special grid $\mathcal{G}$ */
10:         $q_{min} = q_{max} = q_{cur}$;
11:         $q_{max}.(D - 1) = q_{max}.D = 1$;
12:         $q_{min}.(D - 1) = q_{min}.D = \frac{1}{r}$;
13:         /*q_{min} *and* q_{max} *are origin and terminus of 2D space of dimensions* $(D - 1)$ *and* D*/
14:         **if** $Cost(q_{min}) \leq$ CC$_k$ and $Cost(q_{max}) \geq$ CC$_k$ **then**
15:             Augment BCS_k with the output of `2D-FSB` covering a 2D contour of cost $(\beta)^{D-2}$CC$_k$ with cost relaxation factor of β^2;
16:         **end if**
17:     **end for**   /* End of $(D - 2)$ nested **for** loops */
18:     Output BCS_k and set $k = k + 1$; /* Move to next contour */
19: **end while**

20: **Execution Phase:**
21: Run the multi-D `SpillBound` algorithm on the plans corresponding to BCS_i for each contour $\mathcal{IC}_i$;

---

6. EXPERIMENTAL EVALUATION

In this section, we profile the $\gamma - \eta$ performance of `FrugalSpillBound` (FSB) on a representative set of complex OLAP queries, using SB's performance as the reference baseline. The experimental framework is described first and followed by an analysis of the results.

Our test workload is comprised of **21** complex and representative SPJ queries from the TPC-DS benchmark, operating at the base size of 100 GB. The number of relations in the query suite vary from 4 to 10, and a spectrum of join-graph geometries are modeled, including *chain*, *star*, *branch*, etc. Further, we wish to maximize the range of cost values, and hence the number of effective dimensions and contours in the ESS, in order to create the most challenging environments for robustness. This is achieved through an index-rich physical schema that creates indexes on all the attribute columns appearing in the queries.

With the above setup, the initial number of ESS dimensions, comprising of all filter and join predicates, ranged from 5 to 12 dimensions across the queries. Subsequently, after executing the DimRed dimensionality reduction algorithm discussed in Section 5.1, the effective dimensionality came down to a span of 3 to 5

dimensions. As might be expected, the surviving effective dimensions only feature *join predicates*, since the filter predicates were either accurately estimated by the attribute histograms, or had low MSO impact factors and were therefore eliminated by DimRed.

To succinctly characterize the queries, the nomenclature aD_Qb is employed, where a specifies the number of epps, and b the query number in the TPC-DS benchmark. For example, 3D_Q15 indicates TPC-DS Query 15 with three of its predicates considered to be error-prone.

The database engine used in our experiments is a modified version of the PostgreSQL 9.4 engine [18], with the primary additions being: (a) *Selectivity Injection*, required to generate the ESS for SB and the BCS for FSB; (b) *Abstract Plan Execution*, required to force the execution engine to execute a particular plan; (c) *Plan Recosting*, required to cost an abstract plan for a query; and (d) *Time-limited Execution*, required to implement the calibrated sequence of executions with associated time budgets.

6.1 Empirical Validation of APC

We begin with an experimental validation of the APC assumption that is central to the FSB approach. For this purpose, we obtained the cost functions of the POSP plans over the ESS using the selectivity injection feature for all the queries in our evaluation suite. Then, we verified, for each cost function, whether its slope was monotonically non-increasing with selectivity for every 1D projection of the function. Representative results of this evaluation, reflecting 120-plus plans sourced from our query workload, are tabulated in Table 2, for both the constituent PCFs and the aggregate OCS.

In the table, a cell corresponding to OCS (or PCF), under *Average*, captures the % of locations in ESS satisfying the assumption averaged over OCSs (or PCFs) in a query along different projections. Supporting metrics such as *Median*, *Minimum* and *Maximum* are also enumerated to provide a sense of the overall distribution. Note that our FSB approach requires concavity only on the OCS, and the vast majority ($> 95\%$) of locations in the ESS satisfy this slope constraint. Moreover, the median value being 100% for most queries indicates that the majority of OCSs and PCFs do not violate the assumption at all. Further, even the rare violations that surfaced were found to be artifacts of rounding errors, cost-modeling errors, and occasional PCM violations due to the PostgreSQL query optimizer not being entirely cost-based in character.

Table 2: % LOCATIONS IN ESS SATISFYING APC

| Query		Average	Median	Min.	Max.
3D_Q15	OCS	100	100	100	100
	PCF	100	100	100	100
3D_Q96	OCS	100	100	100	100
	PCF	100	100	100	100
4D_Q7	OCS	100	100	100	100
	PCF	98.4	100	74.4	100
4D_Q26	OCS	99.7	100	98.8	100
	PCF	99.7	100	93.9	100
4D_Q27	OCS	100	100	100	100
	PCF	99.2	100	75.2	100
5D_Q19	OCS	100	100	100	100
	PCF	100	100	100	100
5D_Q84	OCS	96.9	96.5	96.5	97.6
	PCF	94.5	96.8	71.2	100
5D_Q91	OCS	100	100	100	100
	PCF	100	100	98.4	100
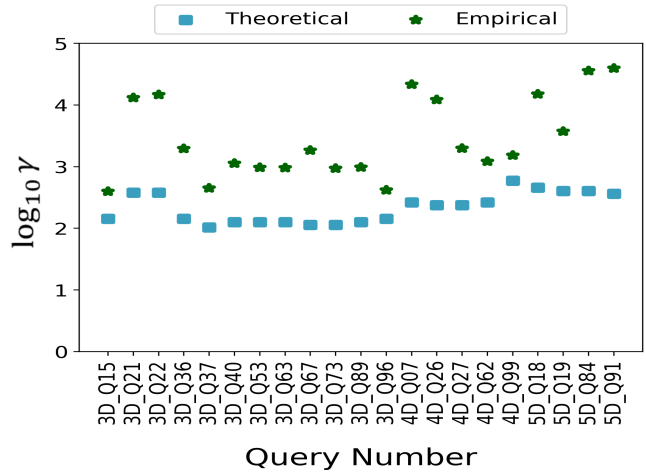
Figure 9: Theoretical Overheads Reduction ($\eta = 2$)

6.2 Theoretical Characterization of $\gamma - \eta$

Using the formula derived in Theorem 5.1, we evaluated the γ value for our suite of benchmark queries with η set to 2, and these results are shown in Figure 9 on a *log scale*. We observe a consistent overheads decrease by more than two orders of magnitude for FSB, i.e. $\gamma >= 100$, over all the queries. Further, the decrease shows a trend of being *magnified* with dimensionality – for instance, the overheads decrease by a factor of almost 400 for the five-dimensional 5D_Q84.

6.3 Empirical Characterization of $\gamma - \eta$

We now turn our attention to assessing the *empirical* reduction in compilation overheads achieved by FSB for the above database environment – these results are also captured in Figure 9. We see here that for most of the queries, the savings are over *three* orders of magnitude. Furthermore, quite a few of the 4D and 5D queries even reach *four* orders of magnitude reduction – in fact, the overheads saving for 5D_Q91 is by a factor of almost 40000! When the effective dimensionality and the number of contours is moderate, as in the case of few 3D queries in Figure 9, the savings become saturated at around 2.5 orders of magnitude since the overheads reach a low value in absolute terms itself, of the order of a few thousand optimization calls.

The reasons for the considerable gap between the theoretical and empirical values include the following:

- Our conservative formulation in Lemma 3.2 for the distances covered by the forward jumps in FSB. These jumps are based on the slope of the optimal plan function at the corresponding location, but the lengths of the jumps in practice are considerably more due to the concave trajectory. For instance, we found that with 5D_Q84, around 60 percent of the jump lengths exceeded 1.5 times the guaranteed value, while about 20 percent were more than twice the guaranteed value.

- Our conservative assumption that all covering contours start from $1/r$ and work their way upto the maximum selectivity of 1. In practice, however, the contour traversals could be much shorter. As a case in point, we found that with 5D_Q84, around 80 percent of the underlying 2D contour explorations were skipped based on the cost condition check in Line 14 of Algorithm 3.
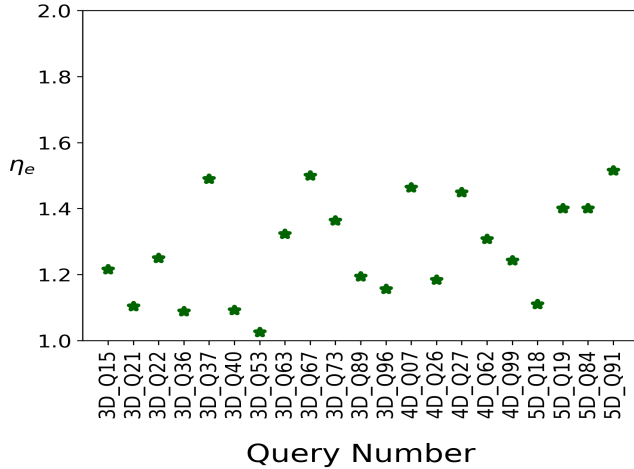
6.4 Validation of MSO Relaxation Constraint

Figure 10: Empirical MSO Ratio ($\eta = 2$)

Figure 11: FSB Tradeoff (Theoretical)

A legitimate concern about FSB could be that while it guarantees maintenance of the η constraint in the theoretical framework, the MSO relaxation may exceed η in the *empirical* evaluation. To assess this possibility, we explicitly evaluated the empirical MSO ratio, η_e, incurred by FSB relative to SB. This was accomplished by exhaustively considering each and every location in the ESS to be q_a, and then evaluating the sub-optimalities incurred for these locations by SB and FSB. Finally, the maximum of these values was taken to represent the empirical MSO of each algorithm.

Contrary to our fears, the η_e values of FSB are always within $\eta = 2$, as shown in Figure 10. In fact, the η_e factors are within 1.5 for all queries. The main reason for the low η_e values in practice is due to the aggressive half-space pruning at each contour, and especially so at the final contour.

6.5 Dependency of γ on η

Thus far, we have analyzed the FSB results for the specific η setting of 2. We now move on to evaluating the γ behavior for different settings of η. This tradeoff is captured in Figure 11 for η values ranging over $[1, 3]$ for three different queries – Q15, Q27 and Q19 – with ESS dimensionalities of 3, 4 and 5, respectively.

We see an initial exponential increase in overheads reduction while going from $\eta = 1$ to $\eta = 2$, but this increase subsequently tapers off for larger values of η. For 3D_Q15, the number of optimization calls decreases steeply from 10^6 to 7010 when η is increased from 1 to 2, and then goes down marginally to 2950 calls when η is further increased to 3. The plateauing of the improvement with increasing η is because a certain minimum number of optimization calls is required for the basic functioning of the FSB algorithm.

6.6 Wall-Clock Time Experiments

All the experiments thus far assessed the $\gamma - \eta$ profile in the abstract world of optimizer cost values. We now present an actual execution experiment, where the end-to-end real-time performance (i.e. wall-clock times) was explicitly measured for the FSB and SB algorithms. Our representative example is based on TPC-DS Q19 featuring 5 error-prone predicates.

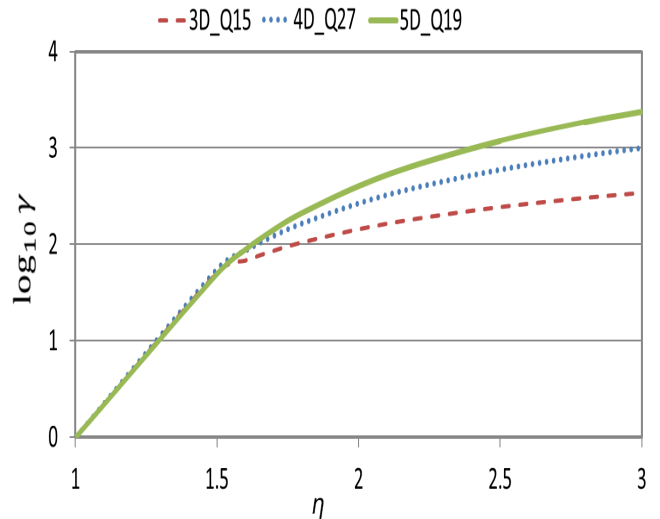As mentioned previously, the task of identifying the contours is inherently amenable to parallelism. Even after exploiting this feature on a 64-core workstation platform, SB took a *few days* to identify all the contours for 5D_Q19. In marked contrast, a parallel version of the BCS identification in FSB, which utilizes the fact that there are $(D*\log_\eta(\frac{1}{\epsilon}))^{D-2}$ independently-explorable $2D$ segments per contour, completed the identification within *10 minutes* (for $\eta = 2$).

After building the ESS, it took SB around 20 mins to complete its query execution, incurring a sub-optimality of 4.8. On the other hand, FSB completed in around 26 mins, resulting in a sub-optimality of 6.2. (The drilled-down information of the specific plan executions for each contour is available in [11].)

So, overall, SB took days to create the ESS and execute this instance of Q19, whereas FSB required only (10 minutes + 26 minutes) = 36 minutes to complete the entire query processing. This means that even if the ad-hoc query eventually turns out to be a canned query, it would take more than 500 successive invocations before SB begins to outperform FSB.

We conducted additional experiments to establish the practicality of the FSB approach. Specifically, on a representative set of queries, we profiled FSB for its memory usage, CPU usage, and end-to-end latency. The memory usage is also a function of the server's database configuration, which was set with the PostgreSQL tuning tool [17]. The results, presented in Table 3, demonstrate that FSB's resource requirements are reasonable and easily justified by the substantive performance benefits that it delivers. Moreover, the CPU usage is relatively small compared to the end-to-end latency since our database environment is disk-bound.

Table 3: RESOURCE USAGE

Query	Memory Usage (MB)	CPU Times (mins)	Latency (mins)
3D_Q15	360	1.4	28.1
3D_Q96	220	1.3	17.8
4D_Q7	489	1.2	23
4D_Q26	490	1.5	12.6
4D_Q27	464	1.8	30.5
5D_Q19	1000	11	36
5D_Q84	348	2.8	10.1
5D_Q91	828	1.3	4.3

7. DEPLOYMENT ASPECTS

Over the preceding sections, we have conducted a theoretical characterization and empirical evaluation of the `FrugalSpillBound` algorithm. We now discuss some pragmatic aspects of its usage in real-world contexts. Most of these issues have already been previously discussed in [12], in the context of the `SpillBound` algorithm, and we therefore only summarize the salient points here for easy reference.

Firstly, we have implicitly assumed a perfect cost model in our study, but this is rarely the case in practice. However, if we were to be assured that the cost modeling errors, while non-zero, are *bounded* within a δ error factor, then the MSO guarantees in this paper will carry through modulo an inflation by a factor of $(1+\delta)^2$. For instance, $\delta = 0.3$ is reported in [1].

Second, it is important to note that both `SpillBound` and `FrugalSpillBound` are *not* substitutes for conventional query optimizers, but are intended to complementarily co-exist with the traditional setup. We currently leave it to the user's discretion about the specific approach to employ for a given query instance – however, we have also begun exploring the use of machine learning techniques to make this determination.

Finally, both `SpillBound` and `FrugalSpillBound` are *intrusive* and require changes to the core engine such as plan spilling and monitoring of operator selectivities. Our experience with PostgreSQL is that these facilities can be incorporated relatively easily. As an aside, the BCS approach can also be used in conjunction with the original `PlanBouquet` algorithm, which operates purely with API functionality.

8. RELATED WORK

In the prior robust query processing literature, there have been two strands of work – the first delivering savings on optimization overheads, and the other addressing the query execution performance – which are discussed in detail below. Given this context, `FrugalSpillBound` appears a unique proposition since it offers an attractive *tradeoff* between these two competing and complementary aspects.

Compilation Overheads. The primary work in this area has been in the context of Parametric Query Optimization (PQO), where the objective is to have *precomputed* the appropriate plans for freshly submitted queries. In [10], the selectivity space was decomposed into polytopes that approximate plan-optimality regions, based on the geometric heuristic that "If all vertices of a polytope share a common optimal plan, then this plan is also optimal *within* the entire polytope". However, this assumption, as well as the presence of regular boundaries for the optimality regions, were later shown in [8] to be largely violated in industrial-strength settings.

Instead of trying to characterize the entire selectivity space in advance, an alternative "pay as you go" approach was taken in [3]. Here, the PQO overheads were restricted only on the actual query workload submitted to the system, facilitating a progressive and efficient exploration of the parameter space. In our setting, however, since we are apriori unaware of the query location, the BCS has to be constructed in an agnostic manner to this location.

More recently, a geometric property called Bounded Cost Growth (BCG) was identified in [6], which typically holds on plan cost functions. In BCG, the relative increase of plan costs is modeled as a low-order polynomial function of the relative increase in plan selectivities. This model was leveraged to ensure bounded sub-optimalities of the PQO choices, relative to the ideal plan at the query location. In fact, using the identity function as the poly-

nomial was itself found to be satisfactory. Our use of concavity is similar to BCG in that, when the polynomial is the identity function, any PCF that satisfies APC also satisfies BCG [11].

Query Execution. The `PlanBouquet` approach [5], based on selectivity discovery instead of estimation, provided guarantees on the worst-case execution performance for the first time. However, its bounds were a function of not only the query, but also the optimizer's behavioral profile over the underlying database platform. `SpillBound` materially extended `PlanBouquet` by providing platform-independent guarantees. Moreover, its empirical performance was markedly superior to `PlanBouquet`. These benefits are due to half-space pruning of the ESS in the discovery process, and bounding the number of plan executions per contour.

Both `PlanBouquet` and `SpillBound` fall under the umbrella of plan-switching approaches. They may therefore appear superficially similar to run-time heuristic re-optimization techniques such as POP [16] and Rio [2]. However, a key difference is their provision of performance guarantees. Further, the heuristic techniques use the optimizer's plan choice as the starting point, and reoptimize at run-time if the estimates are found to be significantly in error. In contrast, `PlanBouquet` and `SpillBound` always start executing plans from the origin of the selectivity space, ensuring repeatability of the query execution strategy as well as controlled switching overheads.

9. CONCLUSIONS

The recently proposed `SpillBound` and `PlanBouquet` discovery-based techniques provide MSO performance guarantees, an essential feature for robust query processing. However, they are only suitable for canned queries due to the enormous compilation overheads incurred prior to initiating query execution.

In this paper, we address the above limitation by designing `FrugalSpillBound` whose compilation overheads are exponentially lower than those of `SpillBound`. Our construction of `FrugalSpillBound` is based on two basic principles: (a) leveraging the axis-parallel concave behavior exhibited by the PCFs and OCS with respect to predicate selectivities in the ESS; (b) substituting the original contours with much smaller contour-covering sets.

Our theoretical analysis establishes a $\eta - \gamma$ tradeoff that is extremely attractive, delivering exponential improvements in γ for linear relaxations in η. Further, the empirical improvements, evaluated on TPC-DS queries, are even higher, by more than an order of magnitude. So, in an overall sense, `FrugalSpillBound` takes an important step towards extending the benefits of MSO guarantees to ad-hoc queries.

`FrugalSpillBound` can intrinsically handle any distributional changes in the data since the entire selectivity range is covered in the ESS. However, handling changes to the database *size* remains an open problem since these updates would result in movement of the iso-cost contours, requiring the bounded contour-covering sets to be discovered from scratch. Therefore, the development of incremental robust algorithms is an interesting future research topic. We also intend to investigate the existence of alternative geometric constraints on cost function behavior – for example, a bounded rate of change – that can be leveraged to further improve the MSO guarantees and/or compilation overheads.

Acknowledgments

10. REFERENCES

[1] Is query optimization a solved problem?
`http://wp.sigmod.org/?p=1075`, 2014.

[2] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proc. of ACM SIGMOD Conf.*, 2005.

[3] P. Bizarro, N. Bruno, and D. DeWitt. Progressive parametric query optimization. *IEEE TKDE*, 21(4):582–594, 2009.

[4] A. Dutt and J. Haritsa. Plan bouquets: Query processing without selectivity estimation. In *Proc. of ACM SIGMOD Conf.*, 2014.

[5] A. Dutt and J. Haritsa. Plan bouquets: A fragrant approach to robust query processing. *ACM TODS*, 41(2):11:1–11:37, 2016.

[6] A. Dutt, V. Narasayya, and S. Chaudhuri. Leveraging re-costing for online optimization of parameterized queries with guarantees. In *Proc. of ACM SIGMOD Conf.*, 2017.

[7] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[8] D. Harish, P. Darera, and J. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.

[9] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *Proc. of 28th VLDB Conf.*, 2002.

[10] A. Hulgeri and S. Sudarshan. Anipqo: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *Proc. of 29th VLDB Conf.*, 2003.

[11] S. Karthik, J. Haritsa, S. Kenkre, and V. Pandit. A concave path to low-overhead robust query processing. Tech. Report TR-2018-01, DSL/CDS, IISc, 2018, `https://tinyurl.com/y9xg7slq`.

[12] S. Karthik, J. Haritsa, S. Kenkre, and V. Pandit. Platform-independent robust query processing. In *Proc. of 32nd IEEE ICDE Conf.*, 2016.

[13] F. Kastrati and G. Moerkotte. Optimization of conjunctive predicates for main memory column stores. *PVLDB*, 9(12):1125–1136, 2016.

[14] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[15] S. Manegold, P. Boncz, and M. Kersten. What happens during a join: Dissecting cpu and memory optimization effects. In *Proc. of 26th VLDB Conf.*, 2000.

[16] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proc. of ACM SIGMOD Conf.*, 2004.

[17] PGTune. `https://pgtune.leopard.in.ua/`.

[18] PostgreSQL. `http://www.postgresql.org/docs/9.4/static/release.html`.

[19] S. Purandare. Dimensionality reduction techniques for bouquet-based approaches. Master's Thesis, Database System Lab, IISc, 2018, `https://tinyurl.com/y97957v7`.

[20] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of ACM SIGMOD Conf.*, 1979.