Platform-independent Robust Query Processing

Srinivas Karthik Jayant Haritsa Sreyash Kenkre[1] Vinayaka Pandit[1]

Technical Report
TR-2015-02

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

[1] IBM Research, India

Abstract

To address the classical selectivity estimation problem in databases, a radically different approach called PlanBouquet was recently proposed in [3], wherein the estimation process is completely abandoned and replaced with a calibrated discovery mechanism. The beneficial outcome of this new construction is that, for the first time, provable guarantees are obtained on worst-case performance, thereby facilitating robust query processing.

The PlanBouquet formulation suffers, however, from a systemic drawback – the performance bound is a function of not only the query, but also the optimizer's behavioral profile over the underlying database platform. As a result, there are adverse consequences: (i) the bound value becomes highly variable, depending on the specifics of the current operating environment, and (ii) it becomes infeasible to compute the value without substantial investments in preprocessing overheads.

In this paper, we present SpillBound, a new query processing algorithm that retains the core strength of the PlanBouquet discovery process, but reduces the bound dependency to only the query. Specifically, SpillBound delivers a worst-case multiplicative bound of $D^2 + 3D$, where D is simply the number of error-prone predicates in the user query. Consequently, the bound value becomes independent of the optimizer and the database platform, and the guarantee can be issued just by inspecting the query, without incurring any additional computational effort.

We go on to prove that SpillBound is within an $O(D)$ factor of the *best possible* deterministic selectivity discovery algorithm in its class. Further, a detailed empirical evaluation over the standard TPC-H and TPC-DS benchmarks indicates that SpillBound provides markedly superior worst-case performance as compared to PlanBouquet in practice. Therefore, in an overall sense, SpillBound offers a substantive step forward in the quest for robust query processing.

1 Introduction

A long-standing problem plaguing database systems is that the predicate selectivity estimates used for optimizing declarative SQL queries are often significantly in error [10, 9]. This results in highly sub-optimal choices of execution plans, and corresponding blowups in query response times. The reasons for such substantial deviations are well documented [15], and include outdated statistics, coarse summaries, attribute-value independence (AVI) assumptions, complex user-defined predicates, and error propagations in the query execution tree. It is therefore of immediate practical relevance to design query processing techniques that limit the deleterious impact of these errors, and thereby provide robust query processing.

We use the notion of Maximum Sub-Optimality (**MSO**), introduced in [3], as a measure of the robustness provided by a query processing technique to errors in selectivity estimation. Specifically, given a query, the MSO of the processing algorithm is the worst-case ratio, over the entire selectivity space, of its execution cost with respect to the optimal cost incurred by an oracular system that magically knows the correct selectivities. It has been empirically determined that MSOs can reach very large values on current database engines [3] – for instance, with Query 19 of the TPC-DS benchmark, it goes as high as a million![2] More importantly, worrisomely large sub-optimalities are *not rare* – for the same Q19, the sub-optimalities for as many as 40% of the locations in the selectivity space were higher than 1000.

As explained in [3], most of the previous approaches to robust query processing (e.g. [10, 1, 12, 8]), including the influential POP and Rio frameworks, are based on heuristics that are not amenable to bounded guarantees on the MSO measure. A notable exception to this trend is the PlanBouquet algorithm, recently proposed in [3], which provides, for the first time, a provable MSO guarantee.

[2] Assuming that estimation errors can range over the entire selectivity space.

Here, the selectivities are not estimated, but instead, systematically *discovered* at run-time through a calibrated sequence of cost-limited executions from a carefully chosen set of plans, called the "plan bouquet". The search space for the bouquet plans is the *Parametric Optimal Set of Plans* (POSP) [6] over the selectivity space. The PlanBouquet technique guarantees $MSO \leq 4 * |PlanBouquet|$. [3]

1.1 PlanBouquet

We describe the working of PlanBouquet with the help of the example query EQ shown in Figure 1, which enumerates orders for cheap parts costing less than 1000. To process this query, current database engines typically estimate three selectivities, corresponding to the two join predicates $(part \bowtie lineitem)$ and $(lineitem \bowtie orders)$, and the filter predicate $(p\_retailprice < 1000)$. While it is conceivable that the filter selectivity may be estimated reliably, it is often difficult to ensure similarly accurate estimates for the join predicates. We refer to such predicates as error-prone predicates, or epp in short (shown bold-faced in Figure 1).

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

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

Example Execution

Given the above query, PlanBouquet constructs a two-dimensional space corresponding to the epps, covering their entire selectivity range ($[0, 1] * [0, 1]$), as shown in Figure 2(a).[4] A location (x, y) in the 2D space corresponds to a scenario in which the selectivities of $(part \bowtie lineitem)$ and $(lineitem \bowtie orders)$ are x and y, respectively. Further, associated with each location in the space are the optimal plan for the location and its execution cost. On this selectivity space, a series of *iso-cost* contours, $\mathcal{IC}_1$ through $\mathcal{IC}_m$, are drawn – each iso-cost contour $\mathcal{IC}_i$ has an associated cost $\mathtt{CC_i}$, and represents the connected selectivity curve along which the cost of the optimal plan(s), as determined by the optimizer, is equal to $\mathtt{CC_i}$. 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 in the following intermediate contours, the cost of each contour is *double* that of the previous contour. That is, $\mathtt{CC_i} = 2^{(i-1)}C$ for $1 < i < m$. The last contour's cost, $\mathtt{CC_m}$, is capped to the maximum query execution cost at the top-right corner of the space.

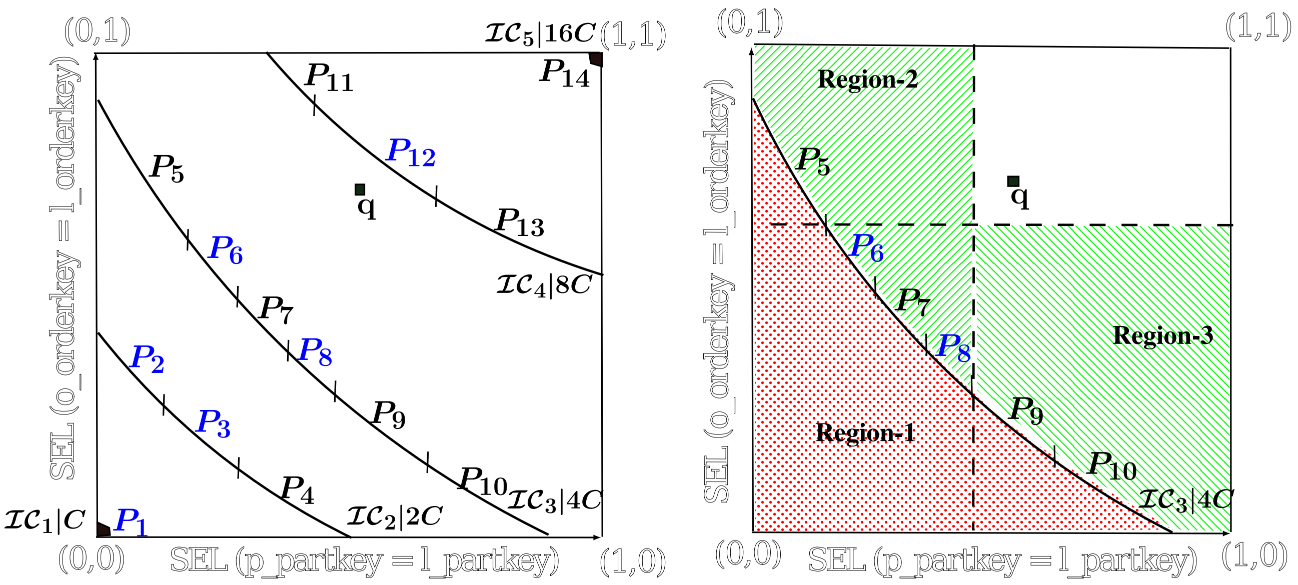
As a case in point, in Figure 2(a), there are five hyperbolic-shaped contours, $\mathcal{IC}_1$ through $\mathcal{IC}_5$, with their costs ranging from C to $16C$. Each contour has a set of optimal plans covering disjoint segments of the contour – for instance, contour $\mathcal{IC}_2$ is covered by plans P_2, P_3 and P_4.

The *union* of the optimal plans appearing on all the contours constitutes the "plan bouquet" – so, in Figure 2(a), plans P_1 through P_{14} form the bouquet. Given this set, the PlanBouquet algorithm operates as follows: Starting with the cheapest contour $\mathcal{IC}_1$, the plans on each contour are sequentially executed *with a time limit equal to the contour's budget*.[5] If a plan fully completes its execution within the assigned time limit, then the results are returned to the user, and the algorithm finishes. Otherwise,

[3] A more precise bound is given later in this section.

[4] Please view the diagrams from a *color copy* to ensure clarity of contents.

[5] We assume a *perfect* cost model, an issue discussed later in this section.



(a) Selectivity Discovery

(b) Pruned Regions - $\mathcal{IC}_3$

Figure 2: `PlanBouquet` and `SpillBound`

as soon as the time limit of the ongoing execution expires, the plan is forcibly terminated and the partially computed results (if any) are discarded. It then moves on to the next plan in the contour and starts all over again. In the event that the entire set of plans in a contour have been tried out without any reaching completion, it *jumps* to the next contour and the cycle repeats.

The basic idea underlying `PlanBouquet` is that it can be shown, under certain mild assumptions, that the first time the (unknown) query location falls within the *hypograph* of a contour, the execution of some plan on the contour is guaranteed to complete the query within the assigned budget. [6] By hypograph we mean the search region *below* the contour curve (after extending, if need be, the corner points of the contour to meet the axes of the search space). A pictorial view is shown in Figure 2(b), which focuses on contour $\mathcal{IC}_3$ – here, the hypograph of $\mathcal{IC}_3$ is the Region-1 marked with red dots.

Now consider the case where the query is located at q, in the intermediate region between contours $\mathcal{IC}_3$ and $\mathcal{IC}_4$, as shown in Figure 2(a). To process this query, `PlanBouquet` would invoke the following budgeted execution sequence:

$$P_1|C, P_2|2C, P_3|2C, P_4|2C, P_5|4C, \ldots, P_{10}|4C, P_{11}|8C, P_{12}|8C$$

with the execution of the final P_{12} plan completing the query.

Performance Guarantees

By sequencing the plan executions and their time limits in the calibrated manner described above, the overheads entailed by this "trial-and-error" exercise can be *bounded*, *irrespective of the query location in the space*. In particular, it is shown that $MSO \leq 4 * \rho$, where ρ is the plan cardinality on the "maximum density" contour. The density of a contour refers to the *number* of plans present on it – for instance, in Figure 2(a), the maximum density contour is $\mathcal{IC}_3$ which features 6 plans.

[6] All points in the ESS fall within the hypograph of at least one contour, and the algorithm is therefore guaranteed to complete the query.

Limitations

The PlanBouquet formulation, while breaking new ground, suffers from a systemic drawback – the specific value of ρ, and therefore the bound, is a function of not only the query, but also the optimizer's behavioral profile over the underlying database platform (including data contents, physical schema, hardware configuration, etc.). As a result, there are adverse consequences: (i) The bound value becomes highly variable, depending on the specifics of the current operating environment – for instance, with TPC-DS Query 25, PlanBouquet's MSO guarantee of 24 under PostgreSQL shot up, under an identical computing environment, to 36 for a commercial engine, due to the change in ρ; (ii) It becomes infeasible to compute the value without substantial investments in preprocessing overheads; and (iii) Ensuring a bound that is small enough to be of practical value, is contingent on the heuristic of "anorexic reduction" [5] holding true.

1.2 SpillBound

The goal of our work is to develop a robust query processing approach that offers an MSO bound which is *solely query-dependent*, irrespective of the underlying database platform. That is, we desire a "structural bound" instead of a "behavioral bound". In this paper, we present a new query processing algorithm, called SpillBound, that materially achieves this objective. Specifically, it delivers an MSO bound that is only a function of D, the *number* of predicates in the query that are prone to selectivity estimation errors. Moreover, the dependency is in the form of a low-order polynomial, with MSO expressed as $(D^2 + 3D)$. Consequently, the bound value becomes:

1. independent of the underlying database platform.[7]
2. known upfront by merely inspecting the query, and not incurring any preprocessing overhead.
3. indifferent to the anorexic reduction heuristic.
4. certifiably low in value for practical values of D.

Example Execution

SpillBound shares the core contour-wise discovery approach of PlanBouquet, but its execution strategy differs markedly. Specifically, it achieves a significant reduction in the cost of the sequence of budgeted executions employed during the selectivity discovery process. For instance, in the example scenario of Figure 2(a), the sequence of budgeted executions correspond to the plans highlighted in blue:

$$P_1|C, P_2|2C, P_3|2C, P_6|4C, P_8|4C, P_{12}|8C$$

with P_{12} again completing the query. Note that the reduced executions result in cost savings of more than 50% over PlanBouquet. The advantages offered by SpillBound are achieved by the follow-

[7] Under the assumption that D remains constant across the platforms.

ing key properties – Half-space Pruning and Contour Density Independent execution – of the algorithm.

Half-space Pruning

PlanBouquet's *hypograph*-based pruning of the selectivity discovery space is extended to a much stronger *half-space*-based pruning. This is vividly highlighted in Figure 2(b), where the half-space corresponding to Region-2 is pruned by the (budget-limited) execution of P_8, while the half-space corresponding to Region-3 is pruned by the (budget-limited) execution of P_6. Note that Region-2 and Region-3 together subsume the entire Region-1 that is covered by PlanBouquet when it crosses $\mathcal{IC}_3$. Our half-space pruning property is achieved by leveraging the notion of *"spilling"*, whereby operator pipelines are prematurely terminated at chosen locations in the plan tree, in conjunction with run-time monitoring of operator selectivities.

Contour Density Independent Execution

In the example scenario, while advancing through the various contours in the discovery process, SpillBound executes at most *two plans* on each contour. In general, when there are D error-prone predicates in the user query, SpillBound is guaranteed to make a *quantum progress* in its discovery process, based on cost-budgeted execution of at most D carefully chosen plans on the contour. Here, a quantum progress refers to a step in which the algorithm either (a) jumps to the next contour, or (b) fully learns the selectivity of some epp (thus reducing the effective number of epps).

Specifically, in each contour, for each dimension, one plan is chosen to be executed in spill-mode (therefore at most D in the contour). The plan chosen for spill-mode execution is the one that provides the *maximal* guaranteed learning of the selectivity along that dimension. In our example, P_8 and P_6 are the two plans chosen for contour $\mathcal{IC}_3$ along the X and Y dimensions, respectively.

1.3 Performance Results

A natural question to ask is whether there might exist some alternative selectivity discovery algorithm, based on half-space pruning, that could provide a much better MSO than SpillBound. In this regard, we theoretically show that *no* deterministic technique in this class can provide an MSO less than D. This result establishes that the SpillBound guarantee is no worse than a factor $O(D)$ in comparison to the best possible algorithm in its class.

The bounds delivered by PlanBouquet and SpillBound are, in principle, *uncomparable*, due to the inherently different nature of their parametric dependencies. However, in order to assess whether the platform-independent feature of SpillBound is procured through a deterioration of the numerical bound, we have carried out a detailed experimental evaluation of both the approaches on standard benchmark queries, operating on the PostgreSQL engine. Moreover, we have empirically evaluated the MSO obtained for each query through an exhaustive enumeration of the selectivity space.

Our experiments indicate that for the most part, SpillBound provides similar guarantees to PlanBouquet, and occasionally, much tighter bounds. As a case in point, for TPC-DS Query 91 with 4 error-prone predicates, the MSO bound is 52.8 with PlanBouquet, but comes down to 28 with SpillBound. More pertinently, the *empirical* MSO of SpillBound is significantly better than that of PlanBouquet for *all* the queries. For instance, the empirical MSO for Q91 decreases drastically from PlanBouquet's 34 to just 7 for SpillBound.

Caveats

While arbitrary selectivity estimation errors are permitted in our study, we have assumed the optimizer's *cost model* to be *perfect* – that is, only optimizer costs are used in the evaluations, and not actual run times. While this assumption is certainly not valid in practice, improving the model quality is, in principle, an orthogonal problem to that of cardinality estimation errors. Dealing with imprecise cost models, and other such practical deployment considerations, are discussed in Section 7.

We hasten to also add that `SpillBound` is *not* a substitute for a conventional query optimizer. Instead, it is intended to complementarily co-exist with the traditional setup, leaving to the user's discretion, the specific approach to employ for a query instance. When small estimation errors are expected, the native optimizer could be sufficient, but if larger errors are anticipated, `SpillBound` is likely to be the preferred choice.

Organization

The remainder of this paper is organized as follows: In Section 2, a precise description of the robust execution problem is provided, along with the associated notations. The building blocks of `SpillBound` are presented in Section 3. The `SpillBound` algorithm and the proof of its MSO bound are presented in Section 4. The lower bound analysis is carried out in Section 5, while the experimental framework and performance results are enumerated 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 Problem Framework

In this section, we present the key concepts, notations, and the formal problem definition. For ease of presentation, we assume that the error-prone selectivity predicates (epps) for a given user query are known apriori, and defer the issue of identifying these epps to Section 7.

2.1 Error-prone Selectivity Space (ESS)

Consider a query with D epps. The set of all epps is denoted by $\texttt{EPP} = \{e_1, \ldots, e_D\}$ 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. In practice, an appropriately discretized grid version of $[0, 1]^D$ is considered as the ESS. Note that each location $q \in [0, 1]^D$ in the ESS represents a specific instance where the epps of the user query happen to have selectivities corresponding to q. Accordingly, the selectivity value on the jth dimension is denoted by $q.j$. We call the location at which the selectivity value in each dimension is 1, i.e, $q.j = 1, \forall j$, as the *terminus*.

The notion of a location q_1 *dominating* a location q_2 in the ESS plays a central role in our framework. Formally, given two distinct locations $q_1, q_2 \in$ ESS, q_1 dominates q_2, denoted by $q_1 \succeq q_2$, if $q_1.j \geq q_2.j$ for all $j \in 1, \ldots, D$. In an analogous fashion, other relations, such as $\not\succeq$, $\preceq$, and $\not\preceq$ can be defined to capture relative positions of pairs of locations.

2.2 Search Space for Robust Query Processing

We assume that the query optimizer can identify the *optimal* query execution plan if the selectivities of all the epps are correctly known.[8] Therefore, given an input query and its epps, the optimal plans for *all* locations in the ESS grid can be identified through repeated invocations of the optimizer with different epp values. The optimal plan for a generic selectivity location $q \in \texttt{ESS}$ is denoted by P_q, and the set of such optimal plans over the complete ESS constitutes the Parametric Optimal Set of Plans (POSP) [6].[9]

We denote the cost of executing an *arbitrary* plan P at a selectivity location $q \in \texttt{ESS}$ by $Cost(P, q)$. Thus, $Cost(P_q, q)$ represents the *optimal* execution cost for the selectivity instance located at q. In this framework, our search space for robust query processing is simply the set of tuples $< q, P_q, Cost(P_q, q) >$ corresponding to all locations $q \in \texttt{ESS}$.

Throughout the paper, we adopt the convention of using q_a to denote the actual selectivities of the user query epps – note that this location is unknown at compile-time, and needs to be explicitly discovered. For traditional optimizers, we use q_e to denote the *estimated* selectivity location based on which the execution plan P_{q_e} is chosen to execute the query. However, this characterization is not applicable to plan switching approaches like PlanBouquet and SpillBound because they explore a *sequence* of locations during their discovery process. So, we denote the deterministic sequence pursued for a query instance corresponding to q_a by $\texttt{Seq}_{q_a}$.

2.3 Maximum Sub-Optimality (MSO) [3]

We now present the performance metrics proposed in [3] to quantify the robustness of query processing.

A traditional query optimizer will first estimate q_e, and then use P_{q_e} to execute a query which may actually be located at q_a. The sub-optimality of this plan choice, relative to an oracle that magically knows the correct location, and therefore uses the ideal plan P_{q_a}, is defined as:

$$SubOpt(q_e, q_a) = \frac{Cost(P_{q_e}, q_a)}{Cost(P_{q_a}, q_a)} \tag{1}$$

The quantity $SubOpt(q_e, q_a)$ ranges over $[1, \infty)$.

With this characterization of a specific (q_e, q_a) combination, the *maximum* sub-optimality that can potentially arise over the entire ESS is given by

$$MSO = \max_{(q_e, q_a) \in \texttt{ESS}} (SubOpt(q_e, q_a)) \tag{2}$$

Further, assuming that all q_a's are equally likely, the *average* sub-optimality (ASO) is given by:

$$ASO = \frac{\sum\limits_{q_a \in \texttt{ESS}} SubOpt(q_e, q_a)}{\sum\limits_{q_a \in \texttt{ESS}} 1} \tag{3}$$

The above definition for a traditional optimizer can be generalized to selectivity discovery algorithms like PlanBouquet and SpillBound. Specifically, suppose the discovery algorithm is currently

[8] For example, through the classical DP-based search of the plan space [14].

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

exploring a location $q \in \texttt{Seq}_{q_a}$ – it will choose P_q as the plan and $Cost(P_q, q)$ as the associated budget. Extending this to the whole sequence, the analogue of Equation 1 is defined as follows:

$$SubOpt(\texttt{Seq}_{q_a}, q_a) = \frac{\sum\limits_{q \in \texttt{Seq}_{q_a}} Cost(P_q, q)}{Cost(P_{q_a}, q_a)} \tag{4}$$

leading to the following MSO and ASO equivalent definitions:

$$MSO = \max_{q_a \in \texttt{ESS}} SubOpt(\texttt{Seq}_{q_a}, q_a) \tag{5}$$

.

$$ASO = \frac{\sum\limits_{q_a \in \texttt{ESS}} SubOpt(\texttt{Seq}_{q_a}, q_a)}{\sum\limits_{q_a \in \texttt{ESS}} 1} \tag{6}$$

2.4 Problem Definition

With the above framework, the problem of robust query processing is defined as follows:

For a given input query Q with its EPP*, and the search space consisting of tuples $< q, P_q, Cost(P_q, q) >$ for all $q \in$* ESS*, develop a query processing approach that minimizes the MSO guarantee.*

As in [3], the primary assumptions made in this paper that allow for systematic construction and exploration of the ESS are those of *plan cost monotonicity* (PCM) and *selectivity independence* (SI). PCM may be stated as: For any two locations $q_b, q_c \in$ ESS, and for any plan P,

$$q_b \succ q_c \Rightarrow Cost(P, q_b) > Cost(P, q_c) \tag{7}$$

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. On the other hand, SI assumes that the selectivities of the epps are all independent – while this is a common assumption in much of the query optimization literature, it often does not hold in practice. In our future work, we intend to look into extending SpillBound to handle the more general case of dependent selectivities.

2.5 Geometric View and Notations

We now present a geometric view of the discovery space and some important notations. Consider the special case of a query with two epps, resulting in an ESS with X and Y dimensions. Now, incorporate a third Z dimension to capture the *cost* of the POSP plans on the ESS, i.e, for $q \in$ ESS, the value of the Z-axis is $Cost(P_q, q)$. This 3D surface, which captures the cost of the POSP plans on the ESS, is called the *Optimal Cost Surface* (OCS). Associated with each point on the OCS is the POSP plan for the underlying location in the ESS. A sample OCS corresponding to the example query **EQ** in the Introduction is shown in Figure 3, which provides a perspective view of this surface. In this figure, the optimality region of each POSP plan is denoted by a unique color. So, for example, the region with blue points corresponds to those locations where the "blue plan" is the optimal plan.[10]

Discretization of OCS: Let C_{min} and C_{max} denote the minimum and maximum costs on the OCS, corresponding to the origin and the terminus of the 3D space, respectively (an outcome of the PCM

[10] Since Figure 3 is only a perspective view of the OCS, it does not capture all the POSP plans.

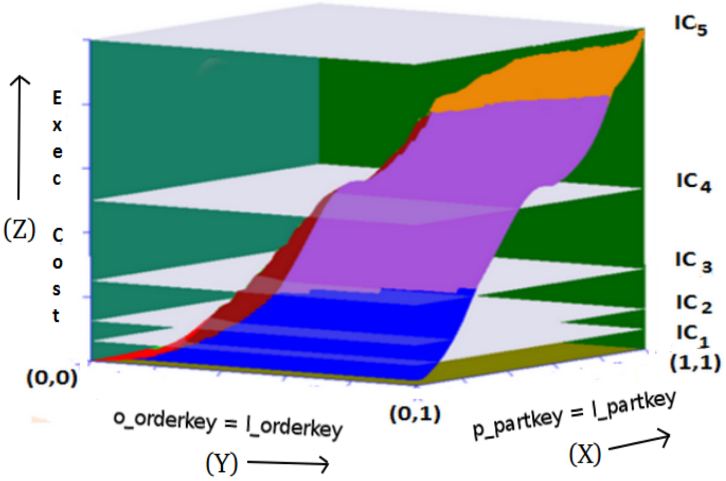


Figure 3: 3D Cost Surface on ESS

assumption). We define $m = \lceil \log_2(\frac{C_{max}}{C_{min}}) \rceil + 1$ hyperplanes that are parallel to the XY plane as follows. The first hyperplane is drawn at C_{min}. For $i = 2, \ldots, m-1$, the i^{th} hyperplane is drawn at $C_{min} \cdot 2^{i-1}$. The last hyperplane is drawn at C_{max}. These hyperplanes correspond to the m isocost contours $\mathcal{IC}_1, \ldots \mathcal{IC}_m$. The isocost contour $\mathcal{IC}_i$ is essentially the 2D curve obtained by intersecting the OCS with the i^{th} hyperplane. We denote the cost of $\mathcal{IC}_i$ by $\mathtt{CC_i}$. The set of plans that are on the 2D curve of $\mathcal{IC}_i$ are referred to as $\mathtt{PL_i}$. For example, in Figure 3, $\mathtt{PL_4}$ includes the purple and maroon plans (in addition to plans that are not visible in this perspective). The *hypograph* of an isocost contour $\mathcal{IC}_i$ is the set of all locations $q \in$ ESS such that $Cost(P_q, q) \leq \mathtt{CC_i}$.

The above geometric intuition and the formal notations readily extend to the general case of D epps, and these notations are summarized in Table 1 for easy reference.

Notation	**Meaning**
epp (EPP)	Error-prone predicate (its collection)
ESS	Error-prone selectivity space
D	Number of dimensions of ESS
$e_1, \ldots, e_D$	The D epps in the query
$q \in [0,1]^D$	A location in the ESS space
$q.j$	Selectivity of q in the jth dimension of ESS
P_q	Optimal Plan at $q \in$ ESS
q_a	Actual run-time selectivity
$Cost(P, q)$	Cost of plan P at location q
$\mathcal{IC}_i$	Isocost Contour i
$\mathtt{CC_i}$	Cost of an isocost contour $\mathcal{IC}_i$
$\mathtt{PL_i}$	Set of plans on contour $\mathcal{IC}_i$

Table 1: Notations

3 Building Blocks of SpillBound

The platform-independent nature of the MSO bound of the SpillBound is enabled by the key properties of half-space pruning and contour density independent execution. In this section, we present these two building blocks of our approach.

3.1 Half-space Pruning

Half-space pruning is the ability to prune half-spaces from the search space based on a single cost-budgeted execution of a contour plan. We now present how half-space pruning is achieved by executing query plans in *spilling mode*. While the use of spilling to accelerate selectivity discovery had been mooted in [3], they did not consider its exploitation for obtaining guaranteed search properties.

We use spilling as the mechanism for modifying the execution of a selected plan – the objective here is to utilize the assigned execution budget to extract increased selectivity information of a specific epp. Since spilling requires modification of plan executions, we shall first describe the query execution model.

3.1.1 Execution Model

We assume the demand driven iterator model, commonly seen in database engines, for the execution of operators in the plan tree [4]. Specifically, the execution takes place in a bottom up fashion with the base relations at the leaves of the tree.

In conventional database query processing, the execution of a query plan can be partitioned into a sequence of *pipelines* [2]. Intuitively, a pipeline can be defined as the maximal concurrently executing subtree of the execution plan. The entire execution plan can therefore be viewed as an ordering on its constituent pipelines. We assume that only one pipeline is executed at a time in the database system, i.e, there is no inter-pipeline concurrency – this appears to be the case in current engines. To make these notions concrete, consider the plan tree shown in Figure 4 – here, the constituent pipelines are highlighted with ovals, and are executed in the sequence $\{L_1, L_2, L_3, L_4\}$.

Finally, we assume a standard plan costing model that estimates the individual costs of the internal nodes, and then aggregates the costs of all internal nodes to represent the estimated cost of the complete plan tree.

3.1.2 Spilling Mode of Execution

We now discuss how to execute plans in spilling mode. For expository convenience, given an internal node of the plan tree, we refer to the set of nodes that are in the subtree rooted at the node as its *upstream* nodes, and the set of nodes on its path to the root as its *downstream* nodes.

Suppose we are interested in learning about the selectivity of an epp e_j. Let the internal node corresponding to e_j in plan P be N_j. The key observation here is that the execution cost incurred on N_j's downstream nodes in P is *not useful* for learning about N_j's selectivity. So, discarding the output of N_j without forwarding to its downstream nodes, and devoting the entire budget to the subtree rooted at N_j, helps to use the budget effectively to learn e_j's selectivity. Specifically, given plan P with cost budget B, and epp e_j chosen for spilling, the spill-mode execution of P is simply the following: Create a modified plan comprised of only the subtree of P rooted at N_j, and execute it with cost budget B.

Since a plan could consist of multiple epps (red coloured nodes in Figure 4), the sequence of spill node choices should be made carefully to ensure guaranteed learning on the selectivity of the chosen node – this procedure is described next.

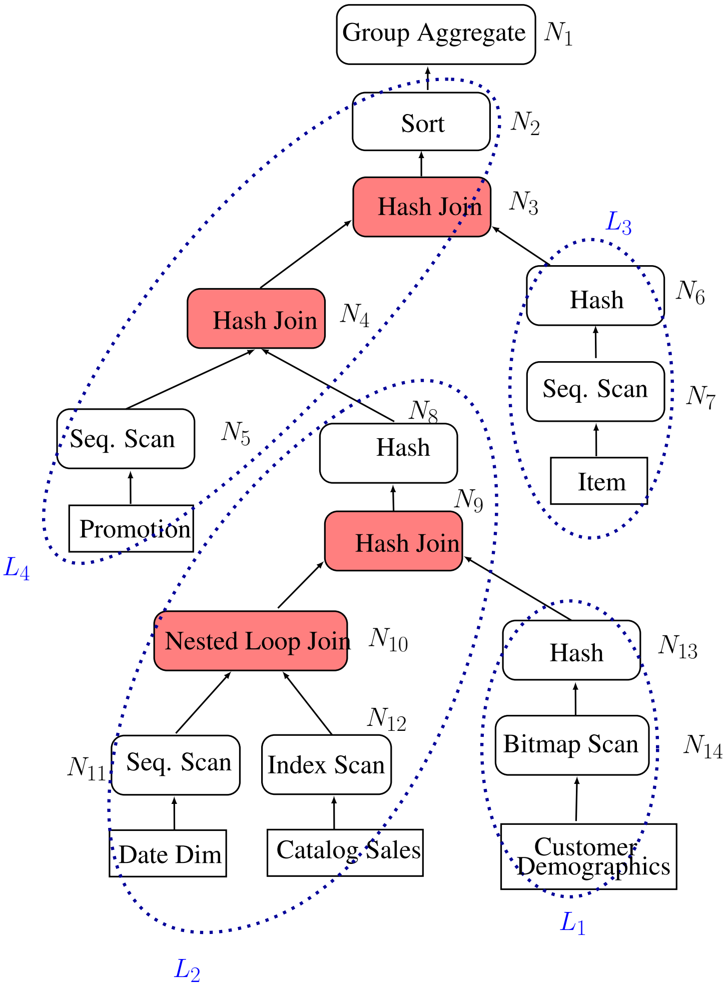


Figure 4: Execution Plan Tree of TPC-DS Query 26

3.1.3 Spill Node Identification

Given a plan and an ordering of the pipelines in the plan, we consider an ordering of epps based on the following two rules:

> *Inter-Pipeline Ordering:* Order the epps as per the execution order of their respective pipelines; in Figure 4, since L_4 is ordered after L_2, the epp nodes N_3 and N_4 are ordered after N_9 and N_{10}.
>
> *Intra-Pipeline Ordering:* Order the epps by their upstream-downstream relationship, i.e., if an epp node N_a is downstream of another epp node N_b within the same pipeline, then N_a is ordered after N_b; in the example, N_3 is ordered after N_4.

It is easy to see that the above rules produce a total-ordering on the epps in a plan – in Figure 4, it is N_{10}, N_9, N_4, N_3. Given this ordering, we always choose to spill on the node corresponding to the *first* epp in the total-order. The selectivity of a spilled epp node is fully learnt when the corresponding execution goes to completion within its assigned budget. When this happens, we remove the epp from EPP and it is no longer considered as a candidate for spilling in the rest of the discovery process.

As a result of this procedure, note that the selectivities of all predicates located *upstream* of the currently spilling epp will be known *exactly* – either because they were never epps, or because they have already been fully learnt in the ongoing discovery process. Therefore, their cost estimates are accurate, leading to the following half-space pruning property.

Lemma 3.1 *Consider a location $q \in$ ESS and the corresponding contour plan P_q. Let* epp *e_j be selected by the spill node identification mechanism. When P_q is executed with budget $Cost(P_q, q)$ and spilling on e_j, then we either learn (a) the exact selectivity of e_j, or (b) that $q_a.j > q.j$.*

Proof 1 *For an internal node N of a plan tree, we use $N.cost$ to refer to the execution cost of the node. Let N_j denote the internal node corresponding to e_j in plan P_q. Partition the internal nodes of P_q into the following: $Upstream(N_j)$, $\{N_j\}$, and $Residual(N_j)$, where $Upstream(N_j)$ denotes the set of internal nodes of P_q that appear before node N_j in the execution order, while $Residual(N_j)$ contains all the nodes in the plan tree excluding $Upstream(N_j)$ and $\{N_j\}$. Therefore, $Cost(P_q, q) = \sum_{N \in Upstream(N_j)} N.cost + N_j.cost + \sum_{N \in Residual(N_j)} N.cost$. The value of the first term in the summation is known with certainty because $Upstream(N_j)$ does not contain any* epp. *Further, the quantity $N_j.cost$ is computed assuming that the selectivity of N_j is $q.j$. Since the output of N_j is discarded and not passed to downstream nodes, the nodes in $Residual(N_j)$ incur zero cost. Thus, when P_q is executed in spill-mode, the budget is sufficiently large to either learn the exact selectivity of e_j (if the spill-mode execution goes to completion) or to conclude that $q_a.j$ is greater than $q.j$.*

Remark. During the entire discovery process of SpillBound, only contour plans are considered for spill-mode executions. Moreover, when we mention the spill-mode execution of a particular plan on a contour, it implicitly means that the budget assigned is equal to the cost of the contour. For ease of exposition, if the epp chosen to spill on is e_j for a plan P, we shall hereafter highlight this information with the notation P^j.

3.2 Contour Density Independent Execution

We now show how the half-space pruning property can be exploited to achieve the contour density independent (CDI) execution property of the SpillBound algorithm. For this purpose, we employ the term "quantum progress" to refer to a step in which the algorithm either jumps to the next contour, or fully discovers the selectivity of some epp. Informally, the CDI property ensures that each quantum progress in the discovery process is achieved by expending no more than |EPP| number of plan executions.

For ease of understanding, we present here the technique for the special case of two epps referred to by X and Y, deferring the generalization for D epps to the next section.

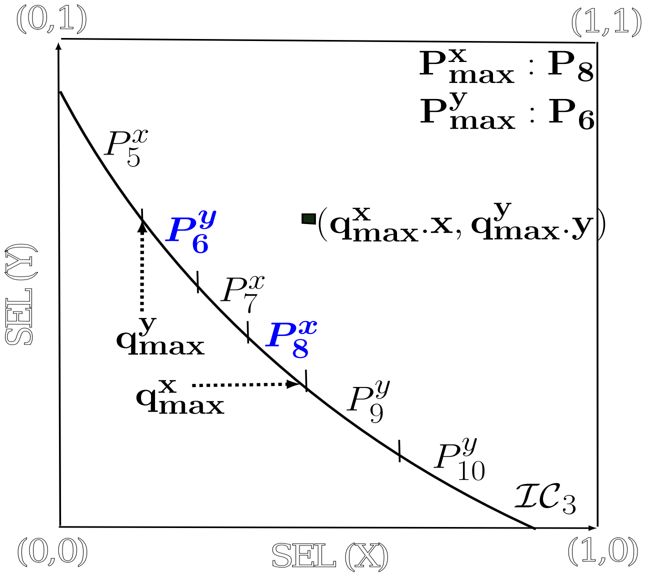


Figure 5: Choice of Contour Crossing Plans

Consider the 2D ESS shown in Figure 5, and assume that we are currently exploring contour $\mathcal{IC}_3$. The two plans for spill-mode execution in this contour are identified as follows: We first identify the subset of plans on the contour that spill on X using the spill node identification algorithm – these plans are identified as P^x_5, P^x_7, P^x_8 in Figure 5. The next step is to enumerate the subset of locations on the contour where these X-spilling plans are optimal. From this subset, we identify the location with the *maximum* X coordinate, referred to as q^x_{max}, and its corresponding contour plan, which is denoted as P^x_{max}. The P^x_{max} plan is the one chosen to learn the selectivity of X – in Figure 5, this choice is P^x_8.

By repeating the same process for the Y dimension, we identify the location q^y_{max}, and plan P^y_{max}, for learning the selectivity of Y – in Figure 5, the plan choice is P^y_6. Note that the location $(q^x_{max}.x, q^y_{max}.y)$ is guaranteed to be either on or beyond the $\mathcal{IC}_3$ contour.

The following lemma shows that the above plan identification procedure satisfies the CDI property.

Lemma 3.2 *In contour $\mathcal{IC}_i$, if plans P^x_{max} and P^y_{max} are executed in spill-mode, and both do not reach completion, then $Cost(P_{q_a}, q_a) >$ CC$_\mathtt{i}$, triggering a jump to the next contour $\mathcal{IC}_{i+1}$.*

Proof 2 *Since the executions of both P^x_{max} and P^y_{max} do not reach completion, we infer that $q^x_{max}.x < q_a.x$ and $q^y_{max}.y < q_a.y$. Therefore, q_a strictly dominates the location $(q^x_{max}.x, q^y_{max}.y)$ whose cost, by PCM, is greater than* CC$_\mathtt{i}$. *Thus $Cost(P_{q_a}, q_a) >$* CC$_\mathtt{i}$.

4 SpillBound Algorithm

In this section, we present our new robust query processing algorithm, SpillBound, which leverages the properties of half-space pruning and CDI execution. We begin by introducing an important notation:

Our search for the actual query location, q_a, begins at the origin, and with each spill-mode execution of a contour plan, we monotonically move closer towards the actual location. The running selectivity location, as progressively learnt by `SpillBound`, is denoted by q_{run}.

For ease of exposition, we first present a version, called `2D-SpillBound`, for the special case of two epps, and then extend the algorithm to the general case of several epps.

4.1 The `2D-SpillBound` Algorithm

To provide a geometric insight into the working of `2D-SpillBound`, we will refer to the two epps, e_1 and e_2, as X and Y, respectively. `2D-SpillBound` explores the doubling isocost contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_m$, starting with the minimum cost contour $\mathcal{IC}_1$. During the exploration of a contour, two plans P^x_{max} and P^y_{max} are identified, as described in Section 3.2, and executed in spill-mode. The order of execution between these two plans can be chosen arbitrarily, and the selectivity information learnt through their execution is used to update the running location q_{run}. This process continues until one of the spill-mode executions reaches completion, which implies that the selectivity of the corresponding epp has been completely learnt.

Without loss of generality, assume that the learnt selectivity is X. At this stage, we know that q_a lies on the line $X = q_a.x$. Further, the discovery problem is reduced to the 1D case, which has a unique characteristic – each isocost contour of the new ESS (i.e. line $X = q_a.x$) contains only *one* plan, and this plan alone needs to be executed to cross the contour, until eventually some plan finishes its execution within the assigned budget. In this special 1D scenario, there is no operational difference between `PlanBouquet` and `2D-SpillBound`, so we simply invoke the standard `PlanBouquet` with only the Y epp, starting from the contour currently being explored. Note that plans are *not* executed in spill-mode in this terminal 1D phase because spilling in the 1D case weakens the bound. This is because, if the plans are executed in spilling mode also in the final 1D phase, this would just lead to learning of the actual selectivity of the left epp. Also since the tuples could be spilled out of the execution plan tree (and not returned to the user), one more execution of a plan at q_a needs to be executed in non-spill mode (regular mode). Thus leading to a bound of one more than what is provided by Theorem 4.2 (this also applies to multidimensional scenario).

4.1.1 Execution Trace

An illustration of the execution of `2D-SpillBound` on TPC-DS Query 91 with two epps is shown in Figure 6. In this example, the join predicate *Catalog Sales* ⋈ *Date Dim*, denoted by X, and the join predicate *Customer* ⋈ *Customer Address*, denoted by Y, are the two epps (both selectivities are shown on a log scale).

We observe here that there are six doubling isocost contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_6$. The execution trace of `2D-SpillBound` (blue line) corresponds to the selectivity scenario where the user's query is located at $q_a = (0.04, 0.1)$.

On each contour, the plans executed by `2D-SpillBound` in spill-mode are marked in blue – for example, on $\mathcal{IC}_2$, plan P_4 is executed in spill-mode for the epp Y. Further, upon each execution of a plan, an axis-parallel line is drawn from the previous q_{run} to the newly discovered q_{run}, leading to the Manhattan profile shown in Figure 6. For example, when plan P_6 is executed in spill-mode for X, the q_{run} moves from (2E-4,6E-4) to (8E-4,6E-4). To make the execution sequence unambiguously clear, the trace joining successive q_{run}s is also annotated with the plan execution responsible for the move – to highlight the spill-mode execution, we use p_i to denote the spilled execution of P_i. So, for instance, the move from (2E-4,6E-4) to (8E-4,6E-4) is annotated with p_6.

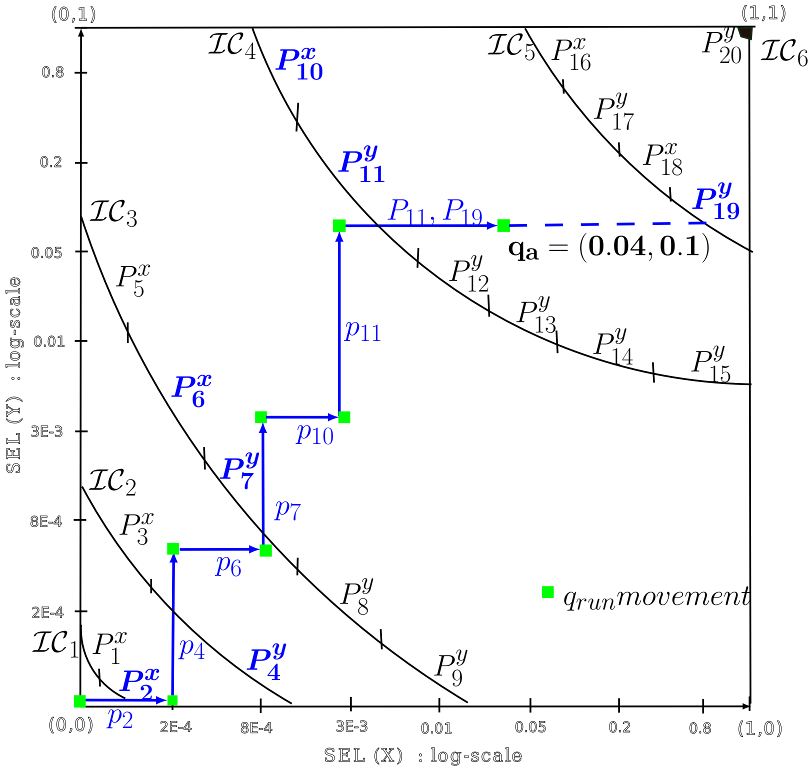


Figure 6: Execution trace for TPC-DS Query 91

With the above framework, it is now easy to see that the algorithm executes the sequence $p_2, p_4, p_6, p_7, p_{10}, p_{11}$, which culminates in the discovery of the actual selectivity of the Y epp. After this, the 1D PlanBouquet takes over and the selectivity of X is learnt by executing P_{11} and P_{19} in regular (non-spill) mode.

This example trace of 2D-SpillBound exemplifies how the benefits of half-space pruning and CDI execution are realized. It is important to note that 2D-SpillBound may execute a few plans *twice* – for example, plan P_{11} – once in spill-mode (i.e., p_{11}) and once as part of the 1D PlanBouquet exploration phase. In fact, this notion of repeating a plan execution during the search process substantially contributes to the MSO bound in the general case of D epps.

4.1.2 Performance Bounds

Consider the situation where q_a is located in the region between $\mathcal{IC}_k$ and $\mathcal{IC}_{k+1}$, or is directly on $\mathcal{IC}_{k+1}$. Then, the 2D-SpillBound algorithm explores the contours from 1 to $k+1$ before discovering q_a. In this process,

Lemma 4.1 *The* 2D-SpillBound *algorithm ensures that at most two plans are executed from each of the contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_{k+1}$, except for one contour in which at most three plans are executed.*

Proof 3 *Let the exact selectivity of one of the* epps *be learnt in contour $\mathcal{IC}_h$, where $1 \leq h \leq k+1$. From CDI execution, we know that* 2D-SpillBound *ensures that at most two plans are executed in each of the contours $\mathcal{IC}_1, \cdots, \mathcal{IC}_h$. Subsequently,* PlanBouquet *begins operating from contour $\mathcal{IC}_h$, resulting in three plans being executed in $\mathcal{IC}_h$, and one plan each in contours $\mathcal{IC}_{h+1}$ through $\mathcal{IC}_{k+1}$.*

We now analyze the worst-case cost incurred by 2D-SpillBound. For this, we assume that the contour with three plan executions is the *costliest* contour $\mathcal{IC}_{k+1}$. Since the ratio of costs between two consecutive contours is 2, the total cost incurred by 2D-SpillBound is bounded as follows:

$$\begin{aligned} TotalCost &\leq 2 * \texttt{CC}_1 + \ldots + 2 * \texttt{CC}_k + 3 * \texttt{CC}_{k+1} \\ &= 2 * \texttt{CC}_1 + \ldots + 2 * 2^{k-1} * \texttt{CC}_1 + 3 * 2^k * \texttt{CC}_1 \\ &= 2 * \texttt{CC}_1 \left(1 + \ldots + 2^k\right) + 2^k * \texttt{CC}_1 \\ &= 2 * \texttt{CC}_1 \left(2^{k+1} - 1\right) + 2^k * \texttt{CC}_1 \\ &\leq 2^{k+2} * \texttt{CC}_1 + 2^k * \texttt{CC}_1 \\ &= 5 * 2^k * \texttt{CC}_1 \end{aligned} \tag{8}$$

From the PCM assumption, we know that the cost for an oracle algorithm (that apriori knows the location of q_a) is lower bounded by $\texttt{CC}_k$. By definition, $\texttt{CC}_k = 2^{k-1} * \texttt{CC}_1$. Hence,

$$MSO \leq \frac{5 * 2^k * \texttt{CC}_1}{2^{k-1} * \texttt{CC}_1} = 10 \tag{9}$$

leading to the theorem:

Theorem 4.2 *The MSO bound of* 2D-SpillBound *for queries with two error-prone predicates is bounded by* **10**.

Remark: Note that even for a ρ value as low as 3, the MSO bound of 2D-SpillBound is better than the $4 * 3 = 12$ offered by PlanBouquet.

4.2 Extending to Higher Dimensions

We now present SpillBound, the generalization of the 2D-SpillBound algorithm to handle D error-prone predicates $e_1, \ldots, e_D$. Before doing so, we hasten to add that the EPP set, as mentioned earlier, is constantly updated during the execution, and epps are removed from this set as and when their selectivities become fully learnt.

The primary generalization that needs to be achieved is to select, prior to exploration of a contour $\mathcal{IC}_i$, the best set (wrt selectivity learning) of $|\texttt{EPP}|$ plans that satisfy the half-space pruning property and ensure complete coverage of the contour. To do so, similar to the 2D case, the plan P^j_{max} corresponding to $e_j \in \texttt{EPP}$ is identified as follows: Among the contour locations for which the corresponding plan spills on e_j, the location with the *maximum* value on the jth coordinate is chosen, and the contour plan at the chosen location is assigned to be P^j_{max}. In essence, among all plans that could provide a guaranteed learning of e_j's selectivity through spill-mode execution, the plan that provides the highest guaranteed learning is chosen.

A subtle but important point to note here is that, *during* the exploration of $\mathcal{IC}_i$, the identity of P^j_{max} *may change* as the contour processing progresses. This is because some of the plans that were assigned to spill on other epps, may switch to spilling on e_j due to their original epps being completely learnt during the ongoing exploration. Accordingly, we term the first execution of a P^j_{max} in contour $\mathcal{IC}_i$ as a *fresh execution*, and subsequent executions on the same epp as *repeat executions*.

Finally, it is possible that a specific epp may have *no* plan on $\mathcal{IC}_i$ on which it can be spilled – this situation is handled by simply skipping the epp. The complete pseudocode for SpillBound is presented in Algorithm 1 – here, Spill-Mode-Execution(P^j_{max},e_j,$\texttt{CC}_\texttt{i}$) refers to the execution of plan P^j_{max} spilling on e_j with budget $\texttt{CC}_\texttt{i}$.

With the above construction, the following lemma can be proved in a manner analogous to that of Lemma 3.2:

Algorithm 1 The `SpillBound` Algorithm

```
Init: i=1, EPP $= \{e_1, \ldots, e_D\}$;
while $i \leq m$ do                                   ▷ for each contour
  if $|$EPP$| = 1$ then                                ▷ only one epp left
    Run PlanBouquet to discover the selectivity of the remaining epp starting from the present contour;
    Exit;
  end if
  Run the spill node identification procedure on each plan in the contour $\mathcal{IC}_i$, i.e, plans in PL_i, and use this information to choose plan $P^j_{max}$ for each epp $e_j$;
  exec-complete = false;
  for each epp $e_j$ do
    exec-complete = Spill-Mode-Execution($P^j_{max}$,$e_j$,CC_i);
    Update $q_{run}.j$ based on selectivity learnt for $e_j$;
    if exec-complete then
      /*learnt the actual selectivity for $e_j$*/
      Remove $e_j$ from the set EPP;
      Break;
    end if
  end for
  if ! exec-complete then
    i = i+1; /* Jump to next contour */
  end if
  Update ESS based on learnt selectivities;
end while
```

Lemma 4.3 *In contour $\mathcal{IC}_i$, if no plan in the set $\{P^j_{max} | e_j \in \texttt{EPP}\}$ reaches completion when executed in spill-mode, then $Cost(P_{q_a}, q_a) > \texttt{CC}_\texttt{i}$, triggering a jump to the next contour $\mathcal{IC}_{i+1}$.*

4.2.1 Performance Bounds

We now present proof of how the MSO bound is obtained for `SpillBound`. In the worst-case analysis of `2D-SpillBound`, the exploration cost of every intermediate contour is bounded by twice the cost of the contour. Whereas the exploration cost of the last contour (i.e., $\mathcal{IC}_{k+1}$) is bounded by three times the contour cost because of the possible execution of a third plan during the `PlanBouquet` phase. We now present how this effect is accounted for in the general case.

Repeat Executions: As explained before, the identity of plan P^j_{max} may dynamically change during the exploration of a contour $\mathcal{IC}_i$, resulting in repeat executions. If this phenomenon occurs, the new P^j_{max} plan would have to be executed to ensure compliance with Lemma 4.3. We observe that each repeat execution of an epp is preceded by an event of fully learning the selectivity of some other epp, leading to the following lemma:

Lemma 4.4 *The `SpillBound` algorithm executes at most D fresh executions in each contour, and the total number of repeat executions across contours is bounded by $\frac{D(D-1)}{2}$.*

Proof 4 *Consider any contour $\mathcal{IC}_i$ for $1 \leq i \leq k+1$. Note that the number of possible fresh executions on contour $\mathcal{IC}_i$ is bounded by D (in fact, it is equal to $|\texttt{EPP}|$ when the algorithm enters the contour $\mathcal{IC}_i$).*

As mentioned earlier, a repeat execution in a contour can happen only when the exact selectivity of one of the `epp`*s is learnt on the contour. Let us say that when the exact selectivity of a* `epp` *is learnt, it marks the beginning of a new phase. If $|\texttt{EPP}|$ is the number of error-prone predicates just before the beginning of a phase, it is easy to see that there are at most $|\texttt{EPP}|-1$ repeat executions within the phase. Further, in each phase the size of* `EPP` *decreases by 1. Therefore, total number of repeat executions is bounded by $\sum_{l=1}^{D-1} l = \frac{D(D-1)}{2}$.* ■

Suppose that the actual selectivity location q_a is located in the range $(\mathcal{IC}_k, \mathcal{IC}_{k+1}]$. Then, the `SpillBound` algorithm explores the contours from 1 to $k+1$ before discovering q_a. Thus, the total cost incurred by the `SpillBound` algorithm is essentially the sum of costs from fresh and repeat executions in each of the contours $\mathcal{IC}_1$ through $\mathcal{IC}_{k+1}$. Further, the worst-case cost incurred by `SpillBound` is when all the repeat executions happen at the costliest contour, $\mathcal{IC}_{k+1}$. Hence, the total cost of the `SpillBound` algorithm is given by

$$\sum_{i=1}^{k+1} (\text{\#fresh executions}(\mathcal{IC}_i)) * \texttt{CC}_\texttt{i} + \tfrac{D(D-1)}{2} * \texttt{CC}_\texttt{k+1} \tag{10}$$

Since the number of fresh executions on any contour is bounded by D, we obtain the following theorem:

Theorem 4.5 *The MSO bound of the* `SpillBound` *algorithm for any query with D error-prone predicates is bounded by $D^2 + 3D$.*

Proof 5 *By substituting the values for no. of fresh executions in each contour by D in equation 10, the total cost for the* `SpillBound` *is*

$$\begin{aligned}
&\leq D * (\sum_{i=1}^{k+1} \texttt{CC}_\texttt{i}) + \frac{D(D-1)}{2} * \texttt{CC}_\texttt{k+1} \\
&= D * (\sum_{i=1}^{k} \texttt{CC}_\texttt{i}) + \frac{D(D+1)}{2} * \texttt{CC}_\texttt{k+1} \\
&= D * (\texttt{CC}_1 + \ldots + 2^{k-1}\texttt{CC}_1) + \frac{D(D+1) * 2^k\texttt{CC}_1}{2} \\
&= D * (2^k - 1)\texttt{CC}_1 + \frac{D(D+1) * 2^k\texttt{CC}_1}{2}
\end{aligned} \tag{11}$$

The cost for an oracle algorithm that a priori knows the correct location of q_a is lower bounded by $2^{k-1}\texttt{CC}_1$. Hence,

$$\begin{aligned}
MSO &\leq \frac{D * (2^k - 1)\texttt{CC}_1 + \frac{D(D+1)*2^k\texttt{CC}_1}{2}}{2^{k-1}\texttt{CC}_1} \\
&\leq 2D + D(D+1) = D^2 + 3D
\end{aligned} \tag{12}$$

■

Remark: Note that the plan located at the end of the principal diagonal in the `ESS` hypercube is guaranteed to ensure the termination of the `2D-SpillBound` and `SpillBound` algorithms for any $q_a \in \texttt{ESS}$.

4.3 Cost Modeling Errors

Thus far, we had assumed that the cost model was perfect but in practice, this is certainly not the case. However, if the modeling errors were to be unbounded, it appears hard to ensure robustness since, in principle, the estimated cost of any plan could be arbitrarily different to the actual cost encountered at run-time. Thus, in a "unbounded estimation errors, bounded modeling errors" framework wherein the modeling errors are non-zero but bounded specifically, the estimated cost of any plan, given correct selectivity inputs, is known to be within a δ error factor of the actual cost. That is, $\frac{c_{estimated}}{c_{actual}} \in [\frac{1}{(1+\delta)}, (1+\delta)]$. Our construction is lent credence to by the recent work of [17], wherein static cost model tuning was explored in the context of PostgreSQL they were able to achieve an average δ value of around 0.4 for the TPC-H suite of queries. This is then amenable to robustness analysis and leads to following result.

Theorem 4.6 *If the cost-modeling errors are limited to error-factor δ with regard to the actual cost, the bouquet algorithm ensures that:*

$$MSO_{bounded\_modeling\_error} \leq MSO_{perfect\_model} * (1+\delta)^2 \tag{13}$$

when $\delta = 0.4$, corresponding to the average in [17], the MSO increases by at most a factor of 2.

5 Lower Bound

In this section, we present a lower bound on the MSO for a class of deterministic half-space pruning algorithms denoted by $\mathcal{E}$. Consider an algorithm $\mathcal{A} \in \mathcal{E}$. Half-space pruning means the following: $\mathcal{A}$ can select an epp j and a plan P, and execute it in such a manner that the selectivity of e_j can be *partly/completely* learnt. Let $PredCost(P, e_j, \ell)$ denote the budget required by an execution of plan P, that allows $\mathcal{A}$ to conclude that $q_a.j > \ell$. For a given epp e_j, we let $CompPredCost(P, e_j)$ denote the minimum budget required by $\mathcal{A}$ to learn the selectivity of e_j completely, using P. Thus an execution of P with budget B to learn e_j allows $\mathcal{A}$ to conclude that

1. $q_a.j$ exceeds ℓ, so that $CompPredCost(P, e_j) > PredCost(P, e_j, \ell)$.
2. $q_a.j$ is at most ℓ, so that $CompPredCost(P, e_j) \leq PredCost(P, e_j, \ell)$; in this case, $q_a.j$ is learned completely.

Note that not all plans P can be used to learn e_j; in this case $PredCost(P, e_j, \ell)$ is ∞, for any $\ell \geq 0$. A spill-mode execution is one of the mechanisms for realizing half-space pruning in practice.

Given a query with an unknown selectivity q_a, the goal of $\mathcal{A}$ is to execute the query to completion. For this, the actions and outcomes of a generic step of $\mathcal{A}$ can be one of the following: (i) a plan P is executed to completion incurring $Cost(P, q_a)$, (ii) a plan P is executed with budget B and it infers that $q \not\succeq q_a$ for all $q \in \texttt{ESS}$ with $Cost(P, q) \leq B$, (iii) a plan P is executed with budget $PredCost(P, e_j, \ell)$, for selectivity j, and learns that (a) $q_a.j > \ell$ or (b) infer $q_a.j$ exactly.

An example of an algorithm that has the capability of executing only (i) and (ii) is PlanBouquet, while SpillBound is an example of an algorithm that has the capability of executing (i), (ii) and (iii). Thus the limitations of the algorithms in $\mathcal{E}$ apply to PlanBouquet and SpillBound. An example of an algorithm that has the capability of executing only (i) above is that of the native optimizer.

Notion of Separation: For a given $q \in \texttt{ESS}$, we let $\mathcal{A}(q)$ denote the sequence of steps taken by $\mathcal{A}$, when the unknown point q_a is q. A convenient way of describing $\mathcal{A}(q_a)$, i.e. the execution of $\mathcal{A}$, is by keeping track of the regions of the ESS where q_a is likely to be. At any step of its execution, if the action performed by $\mathcal{A}$ is hypograph pruning (action (ii)) or half space pruning (action (iii)), then it rejects certain locations in the ESS as possible q_a locations. At the completion of step t, we let $W_t^{q_a}$ be the set of locations of the ESS which are *not* pruned by $\mathcal{A}$, and let T^{q_a} be the total number of steps performed by $\mathcal{A}(q_a)$. Thus $W_0^{q_a} = \texttt{ESS}$, and we describe $\mathcal{A}(q_a)$ to be the sequence $W_0^{q_a}, W_1^{q_a}, \ldots, W_{T^{q_a}}^{q_a}$. Hence we can view the execution of $\mathcal{A}$ as a sequence of steps in which locations of the ESS are separated out from the unknown q_a, until the query is successfully executed. Note that $\mathcal{A}$ need not explicitly maintain the $W_t^{q_a}$; it is simply a means of describing the execution of $\mathcal{A}$.

We say that $\mathcal{A}(q_a)$ *separates* $q_1, q_2 \in \texttt{ESS}$ if at some step t in its execution, $q_1 \in W_t^{q_a}$, $q_2 \notin W_t^{q_a}$, while q_1, q_2 were both in $W_{t-1}^{q_a}$. More generally, for two disjoint subsets of the ESS, U_1 and U_2, we say that $\mathcal{A}(q_a)$ *separates* the set $U_1 \cup U_2$ into U_1 and U_2, if there is a step t such that $U_1 \cup U_2 \subseteq W_{t-1}^{q_a}$, but $U_1 \subseteq W_t^{q_a}$ and $U_2 \cap W_t^{q_a} = \phi$ (i.e. U_1 is a subset of $W_t^{q_a}$, while U_2 is disjoint from $W_t^{q_a}$).

Consequence of Deterministic Behavior: The algorithms we consider are deterministic. Thus the action of $\mathcal{A}$ at a step is determined completely by the actions and outcomes of previous steps. A formal way to capture this is as follows. Let q_1 and q_2 be two points of the ESS. Let t be the largest number such that $q_2 \in W_t^{q_1}$, and t' be the largest number such that $q_1 \in W_{t'}^{q_2}$. Since $W_0^{q_1} = W_0^{q_2} = \texttt{ESS}$, these points exist. At $min(t, t')$, and $W_i^{q_1} = W_i^{q_2}$ for $i = 0, 1, \ldots, t$. We are now ready to prove the lower bound.

Theorem 5.1 *For any algorithm $\mathcal{A} \in \mathcal{E}$ and $D \geq 2$, there exists a D-dimensional* ESS *where the MSO of $\mathcal{A}$ is at least D.*

Construction of ESS: Suppose the MSO of $\mathcal{A}$ is strictly less than D. We construct a special D-dimensional search space on which the contradiction is shown. It is constructed with the help of a set of locations $V = \{q_1, \ldots, q_D\}$ given by $q_i.j = 1/D$ if $j = i$, else $q_i.j = 1$. Further, our construction is such that the ESS will have exactly D plans $P_1, P_2, \ldots, P_D$. The cost structure is as follows:

$$\begin{aligned} Cost(P_i, q) &= D * q.i \; \forall q \in \texttt{ESS} \\ Cost(P_i, q.j) &= D * q.j \; \forall q \in \texttt{ESS}, \texttt{epp}\ j \end{aligned}$$

Thus the POSP plan at q_i is P_i and has a cost of 1. For a two dimensional ESS and a cost c, the iso-cost curves correspond to L shaped objects, consisting of two segments, *blue* and *red*, as shown in the figure 7. The blue segments consist of all points q with $q.x = c/2$, and $q.y \geq c/2$. Similarly, the red segments consist of all points q with $q.y = c/2$, and $q.x \geq c/2$. The points q_1 and q_2 correspond to $(1/2, 1)$ and $(1, 1/2)$ respectively.

We verify the PCM property as follows. For a plan P_j, if $q_1 \preceq q_2$, then $q_1.j \leq q_2.j$; then $Cost(P_j, q_1) = D * q_1.j \leq D * q_2.j \leq Cost(P_j, q_2)$. Note that we have allowed equality in the definition of the PCM for ease of exposition. We explain the proof with this relaxed version of the PCM, and in the last part of this section we show a modification to the costs that allows the same proof to work for the strict version of the PCM property.

Claim 5.1 *Let $q_a \in V$. Let V_1, V_2 be such that $V_1 \cap V_2 = \phi$ and $V_1 \cup V_2 = V_3 \subseteq V$. If $\mathcal{A}$ separates V_3 into V_1 and V_2, then either $|V_1| = 1$ or $|V_2| = 1$.*

If the claim is false, then $\mathcal{A}(q_a)$ splits V_3 into V_1 and V_2 each of size at least two. Let q_{i_1}, q_{i_2} and q_{i_3}, q_{i_4} be the locations in V_1 and V_2 respectively. Then $\mathcal{A}$ separates q_{i_1}, q_{i_2} from q_{i_3}, q_{i_4} in the same step. By the conditions on $\mathcal{A}$, at least one of the following must have happened.

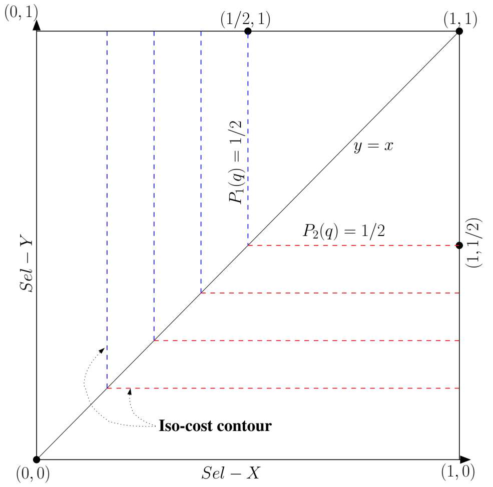


Figure 7: ESS for Theorem 5.1

1. $\mathcal{A}$ explores a location q and concludes that q_{i_1}, q_{i_2} both $\prec q$, while $q_{i_3}, q_{i_4} \not\prec q$ (or vice-versa, in which case interchange the roles of V_1 and V_2). By construction of V, if q_{i_1}, q_{i_2} both $\prec q$, then q has to be such that $q.j = 1\ \forall j \in 1, \ldots, D$, i.e, $q = \mathbf{1}$. But, this implies that $q_{i_3}, q_{i_4} \preceq q$ (contradiction).

2. $\mathcal{A}$ identifies an `epp` j, a plan P and budget B such that $q_{i_1}.j, q_{i_2}.j$ are learned, while $q_{i_3}.j, q_{i_4}.j$ cannot be learned within budget B. Since $i_1 \neq i_2$, the budget utilized for learning the selectivities is at least D. Since $q_a \in V$, its POSP cost is 1. So, the MSO of $\mathcal{A}$ is at least D (contradicting the assumption that MSO is less than D).

This proves the above claim. From the above, we see that to split V_3, $\mathcal{A}$ needs a cost of at least 1. We are now ready to prove the Theorem 5.1.

Proof 6 *(of Theorem 5.1) Suppose $\mathcal{A} \in \mathcal{E}$ has an MSO less than D. The POSP plan at $q_i \in V$ is P_i, and it incurs a cost of 1 to execute. The cost of executing P_i at $q_j \in V$, where $j \neq i$ is D. Since the MSO of $\mathcal{A}$ is less than D, the final step of $\mathcal{A}(q_i)$ cannot be the same for two different q_i, q_j. Thus the execution of $\mathcal{A}(q_i)$ and $\mathcal{A}(q_j)$ differs and $\mathcal{A}$ separates q_i and q_j. Choose q_a arbitrarily from $V_0 = V$ and execute $\mathcal{A}$. Consider the step in which $\mathcal{A}$ separates V_0 the first time. Suppose q_1 is separated from $V_1 = V_0 \setminus \{q_1\}$ in this step. Then choose q_a arbitrarily from V_1, and execute $\mathcal{A}(q_a)$ again. Since $\mathcal{A}$ is deterministic, $\mathcal{A}(q_1)$ and $\mathcal{A}(q_a)$ are identical till V_0 is first separated. Thus, it will first separate V_0 and then V_1. Suppose it separates q_2 from $V_2 = V_1 \setminus \{q_2\}$. Choose q_a arbitrarily from V_2, and execute $\mathcal{A}(q_a)$ again. It will first separate V_0, then V_1, and then V_2. Suppose it separates q_3 from $V_3 = V_2 \setminus \{q_3\}$. Choose q_a arbitrarily from V_3 and repeat this process inductively. Say q_D is left at the starting of Dth step, then $q_a = q_D$, $\mathcal{A}$ separates each of $V_0, V_1, \ldots, V_{D-1}$ in different steps, and finally complete q_a successfully. As each separation step needs a cost of at least 1, and a cost of at least 1 to execute q_a, $\mathcal{A}$ pays a cost of at least D for $q_a = q_D$. But, the cost of P_D at q_D is 1. Thus, the MSO of $\mathcal{A}$ is at least D, which contradicts our assumption.*

We thus have the following corollary.

Corollary 5.2 *For $D \geq 2$, there exists an* ESS*, where any deterministic half-space pruning based algorithm has an MSO of at least D*

Dealing with strict PCM: The strict PCM property is as follows: if q_1 and q_2 are two points of the ESS such that $q_1 \prec q_2$, then for all plans P, $Cost(P, q_1) < Cost(P, q_2)$. The cost function we constructed above does not satisfy this property. However, the following cost functions follow the strict PCM property. The plans are $P_1, \ldots, P_D$ as before. Their cost structure is now as follows.

$$\begin{aligned}\overline{Cost}(P_i, q) &= D * q.i + \delta \sum_{j \neq i} q.j \ \forall q \in \texttt{ESS} \\ \overline{Cost}(P_i, q.j) &= D * q.j \ \forall q \in \texttt{ESS}, \texttt{epp}\ j\end{aligned}$$

In the above δ is a very small positive constant whose exact value is chosen based on what we are trying to prove. Note that since the cost function is a sum of increasing linear terms, the full function is an increasing linear function.

Claim 5.2 *The above cost function $\overline{Cost}(.,.)$ obey the strict PCM property.*

Proof 7 *Let q_1 and q_2 be points in the* ESS *such that $q_1 \prec q_2$. Since the cost function corresponding to any plan P_i are increasing, we have*

$$D(q_1.i) + \delta \sum_{j \neq i} q_1.j \quad \leq \quad D(q_2.i) + \delta \sum_{j \neq i} q_2.j$$

So that

$$D(q_2.i - q_1.i) + \delta \sum_{j \neq i} (q_2.j - q_1.j) \quad \geq \quad 0$$

Since $q_1 \prec q_2$, the above is a sum of non-negative terms. Since the relation is strict, there is at least one k in $1, \ldots, D$, such that $q_1.k < q_2.k$, the above sum is strictly greater than zero.

Note that $\overline{Cost}(P_i, q_i) = 1 + \delta(D-1)$, and $\overline{Cost}(P_i, q_j) = D + \delta(D - 2 + 1/D)$. We then modify the above theorem as follows.

Theorem 5.3 *For any algorithm $\mathcal{A} \in \mathcal{E}$ and $\epsilon > 0$, for every D, there is a D-dimensional* ESS *where the MSO of $\mathcal{A}$ is at least $D - \epsilon$.*

To prove the above theorem, we note the following. Let $U \subseteq V$, and $q_i \in U$ be such that $\mathcal{A}$ separates q_i from $U \setminus \{q_i\}$. Then either $\mathcal{A}$ discovers a point q such that $q_i \preceq q$ while it does not dominate any point of U or vice versa. This means that q dominates some point of V. So the cost of executing a plan at q is at least $1 + \delta(D-1)$ which exceeds 1. Thus, to separate any two points of V, a cost of at least 1 is required.

Now suppose $\mathcal{A}$ has an MSO of at most $D - \epsilon$ for some $\epsilon > 0$. Take $\delta = \frac{\epsilon}{D^2 - 1}$. Then it is easy to verify that $\overline{Cost}(P_i, q_j)/\overline{Cost}(P_i, q_i)$ exceeds $D - \epsilon$. So, the final step of $\mathcal{A}(q_i)$ cannot be the same for two different q_i, q_j. We now proceed on similar lines to the proof of Theorem 5.1.

6 Experimental Evaluation

As mentioned earlier, the bounds delivered by `PlanBouquet` and `SpillBound` are not directly comparable, due to the inherently different nature of their dependencies on the ρ and D parameters, respectively. However, we need to assess whether the platform-independent feature of `SpillBound` is procured at the expense of a deterioration in the numerical bounds. Accordingly, we present in this section an evaluation of `SpillBound` on a representative set of complex OLAP queries, and compare its worst-case performance (MSO) with `PlanBouquet`. The experimental framework, which is similar to that used in [3], is described first, followed by an analysis of the results.

6.1 Database and System Framework

Our test workload is comprised of representative SPJ queries from the TPC-DS and TPC-H benchmarks operating at their base sizes of 100GB and 1GB, respectively. The number of relations in these queries range from 4 to 10, and a spectrum of join-graph geometries are modeled, including *chain*, *star*, *branch*, etc. The number of epps range from 2 to 6, all corresponding to join predicates, giving rise to challenging multi-dimensional ESS spaces. We will now present the results for TPC-DS queries and TPC-H queries.

To succinctly characterize the queries, the nomenclature xD_y_Qz is employed, where x specifies the number of epps, y the benchmark (H or DS) and z the query number in the benchmark. For example, 3D_DS_Q15 indicates Query 15 of TPC-DS benchmark with three of its join predicates considered to be error-prone .

The database engine used in our experiments is a modified version of the PostgreSQL 8.4 [13] engine, with the primary changes being the incorporation of spilling and time-limited execution of plans. Due to the intrusive nature of spilling, we have not shown experimental results on commercial database engines.

The MSO guarantee for PlanBouquet on the original ESS typically turns out to be very high due to the large values of ρ. Therefore, as in [3], we conduct the experiments for PlanBouquet only after carrying out the *anorexic reduction* transformation [5], using the default $\lambda = 20\%$ replacement threshold – we use ρ_{RED} to refer to this reduced value.

In the remainder of this section, for ease of exposition, we use the abbreviations PB and SB to refer to PlanBouquet and SpillBound, respectively. Further, we use *MSO$_g$* (MSO guarantee) and *MSO$_e$* (MSO empirical) to distinguish between the MSO guarantee and the empirically evaluated MSO obtained on our suite of queries.

6.2 Comparison of MSO guarantees (MSO$_g$)

A summary comparison of MSO$_g$ for PB and SB over almost a dozen TPC-DS and TPC-H queries of varying dimensionality is shown in Figure 8 – for PB, they are computed as $4(1+\lambda)\rho_{RED}$, whereas for SB, they are computed as $D^2 + 3D$.

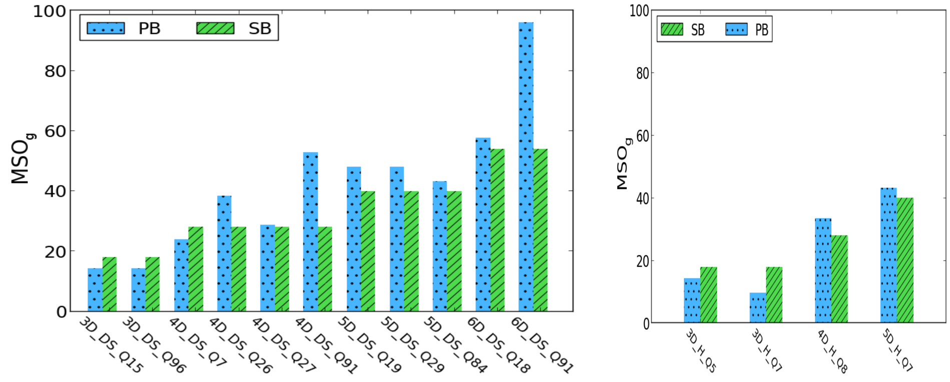
We observe here that in a few instances, specifically 4D_DS_Q26 and 4D_DS_Q91, SB's guarantee is noticeably *tighter* than that of PB – for instance, the values are 28 and 52.8, respectively, for 4D_DS_Q91. In the remaining queries, the bound quality is roughly similar between the two algorithms with only marginal differences. Therefore, contrary to our fears, the MSO guarantee has not suffered due to incorporating platform independence.

6.3 Variation of MSO Guarantee with Dimensionality

In our next experiment, we investigated the behavior of MSO$_g$ as a function of ESS dimensionality for a given query. We present results here for an example TPC-DS query, namely Query 91, wherein the number of epps were varied from 2 upto 6. The results are presented in Figure 9. We observe here that while SB is marginally worse at the lowest dimensionality of 2, it becomes appreciably better than PB with increasing dimensionality – in fact, at 6D, the values are 96 and 54 for PB and SB, respectively.

6.4 Comparison of Empirical MSO (MSO$_e$)

The previous experiments focused on characterizing the MSO bounds. We now move on to evaluating the empirical MSO, MSO$_e$, incurred by the two algorithms. There are two reasons that it is important to



(a) TPC-DS

(b) TPC-H

Figure 8: Comparison of MSO guarantees (MSO$_g$)

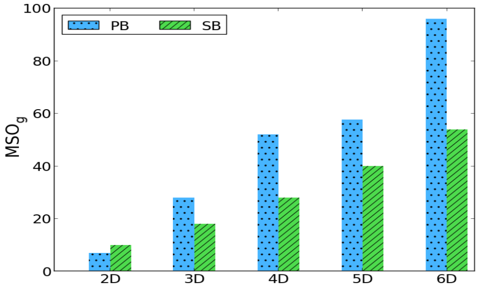


Figure 9: Variation of MSO$_g$ with Dimensionality (DS_Q91)

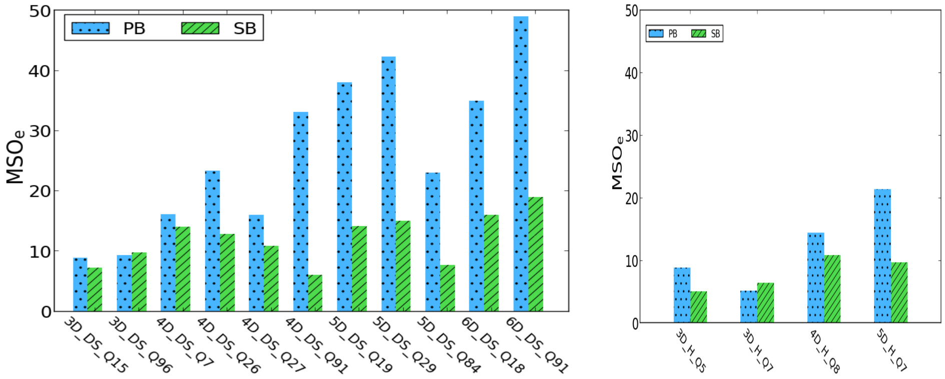
carry out this exercise: Firstly, to evaluate the looseness of the bounds. Secondly, to evaluate whether PB, although having weaker bounds in theory, provides better performance in practice, as compared to SB.

The assessment was accomplished by explicitly and exhaustively considering each and every location in the ESS to be q_a, and then evaluating the sub-optimality incurred for this location by PB and SB. Finally, the maximum of these values was taken to represent the MSO$_e$ of the algorithm.

The MSO$_e$ results are shown in Figure 10 for the entire suite of test queries. Our first observation is that the empirical performance of SB is far better than the corresponding bound values shown in Figure 8. In contrast, while PB also shows improvement, it is not as dramatic. For instance, considering 6D_DS_Q18, PB reduces its MSO from 57.6 to 35.2, whereas SB reduces from 54 to just 16.

The second observation is that the gap between SB and PB is *accentuated* here, with SB performing substantially better over a larger set of queries. For instance, consider query 5D_DS_Q29, where the MSO$_g$ values for PB and SB were 52.8 and 40, respectively – the corresponding empirical values are 42.3 and 15.1 in Figure 10.

Finally, even for a query like 4D_DS_Q7, where PB had a marginally *better* bound (24 for PB and 28 for SB in Figure 8), we find that it is now SB which is found to be superior in practice (16.1 for PB and 13.9 for SB in Figure 10).



(a) TPC-DS

(b) TPC-H

Figure 10: Comparison of Empirical MSO (MSO$_e$)

6.5 Analysis of Looseness of SB's *MSO*$_g$

We now profile the execution of the queries to investigate the significant gap between SB's MSO$_g$ and MSO$_e$ values. Recall that the analysis (Section 4.2.1) bounded the cost of repeat executions by attributing *all of them* to the last contour, i.e., $\mathcal{IC}_{k+1}$. Moreover, the number of fresh executions in all the contours, including $\mathcal{IC}_{k+1}$, was assumed to be D. This results in the execution cost over $\mathcal{IC}_{k+1}$ being the dominant contributor to MSO$_g$. To quantitatively assess this contribution, we present in Tables 2 and 3 the drilled-down information of: (i) the number of fresh executions of plans on $\mathcal{IC}_{k+1}$, and (ii) the number of repeat executions of plans on $\mathcal{IC}_{k+1}$. For each of these factors, we present both the theoretical and empirical values. Note that the specific q_a locations used for obtaining these numbers corresponds to the locations where the MSO was empirically observed.

Armed with the statistics of Tables 2 and 3, we conclude that the main reasons for the gap are the following: Firstly, while the number of repeat executions in contour $\mathcal{IC}_{k+1}$, as per the analysis, is $D(D-1)/2$, the empirical count is far fewer – in fact there are *no* repeat executions in queries such as 3D_DS_Q96, 4D_DS_Q7, 4D_DS_Q27, 4D_DS_Q91 and 5D_DS_Q19. While it is possible that repeat executions did occur in the earlier lower cost contours, their collective contributions to sub-optimality are not significant.

Secondly, by the time the execution reaches the $\mathcal{IC}_{k+1}$ contour, it is likely that the selectivities of some of the epps have *already been learnt*. The bound however assumes that *all* selectivities are learnt only in the last contour. As a case in point, for 5D_DS_Q19, the selectivities of three of the five epps had been learnt prior to reaching the last contour. In fact the number of epps left in contour $\mathcal{IC}_{k+1}$ is equal to the number of fresh executions in $\mathcal{IC}_{k+1}$.

6.6 Average-case Performance (ASO)

Apart from MSO, we also evaluated the average sub-optimality (ASO), as defined in Equation 6, of PB and SB over the entire ESS for all the test queries. These results are shown in Figure 11. Again, we observe that, just as in the case of MSO, SB provides either similar performance, or much better, especially at higher dimensions, as compared to PB.

It can be seen that as the number of dimensions increases, the average case performance of SB is much better than PB. Thus, our approach offers significant benefits over PlanBouquet both in terms of worst-case and average-case behavior.

Query	**Fresh Executions in $\mathcal{IC}_{k+1}$**		**Repeat Executions in $\mathcal{IC}_{k+1}$**	
	Bound	Empirical	Bound	Empirical
3D_DS_Q15	3	2	3	1
3D_DS_Q96	3	2	3	0
4D_DS_Q7	4	3	6	0
4D_DS_Q26	4	4	6	4
4D_DS_Q27	4	4	6	0
4D_DS_Q91	4	3	6	0
5D_DS_Q19	5	2	10	0
5D_DS_Q29	5	4	10	2
5D_DS_Q84	5	3	10	1
6D_DS_Q18	6	4	15	1
6D_DS_Q91	6	6	15	7

Table 2: TPC-DS: Sub-optimality Contribution of IC_{k+1}

Query	**Fresh Executions in $\mathcal{IC}_{k+1}$**		**Repeat Executions in $\mathcal{IC}_{k+1}$**	
	Bound	Empirical	Bound	Empirical
3D_H_Q5	3	2	3	1
3D_H_Q7	3	2	3	1
4D_H_Q8	4	3	6	2
5D_H_Q7	5	3	10	2

Table 3: TPC-H: Sub-optimality Contribution of IC_{k+1}

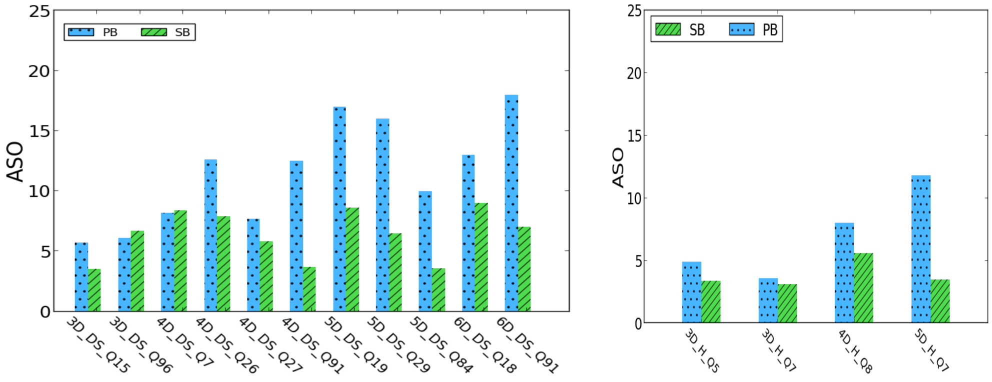
6.7 Sub-Optimality Distribution

In our final analysis, we profile the *distribution* of sub-optimality over the ESS. That is, a histogram characterization of the number of locations with regard to various sub-optimality ranges. A sample histogram, corresponding to query 4D_DS_Q91, is shown in Figure 12, with sub-optimality ranges of width 5. We observe here that for over 90% of the ESS locations, the SO of SB is less than 5. Whereas this performance is achieved for only 35% of the locations using PB. Similar patterns were observed for the other queries as well, and these results indicate that from both *global and local* perspectives, SB has desirable performance characteristics as compared to PB.

We present the results for query 4D_DS_Q91. For this query, the sub-optimality of SB for over 90% of the locations was less than 5. In comparison, the sub-optimality was less than 5 for only 35% of the locations in case PB. In Figure 12, we show the % of locations that have different range of sub-optimality over the entire ESS. We would like to highlight that similar patterns were observed for other test queries as well.

7 Deployment Aspects

Over the preceding sections, we have conducted a theoretical characterization and empirical evaluation of the SpillBound algorithm. We now discuss some pragmatic aspects of its usage in real-world



(a) TPCDS

(b) TPCH

Figure 11: Comparison of ASO performance

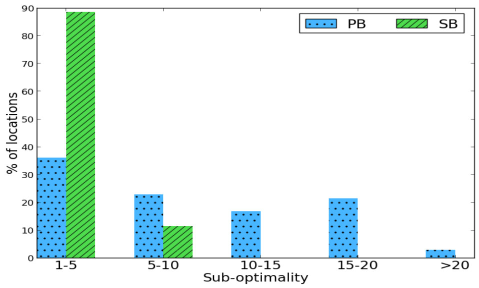


Figure 12: Sub-optimality Distribution (4D_DS_Q91)

contexts. Most of these issues have already been previously discussed in [3], in the context of the `PlanBouquet` algorithm, and we summarize the salient points here for easy reference.

First, our assumption of a perfect cost model. 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 $(1+\delta)^2$. That is, the MSO guarantee of `SpillBound` would be $(D^2+3D)(1+\delta)^2$ as mentioned in Section 4.3.

Second, with regard to identification of the epps that constitute the ESS, we could leverage application domain knowledge and query logs to make this selection, or simply be conservative and assign all uncertain predicates to be epps.

Third, the construction of the contours in the ESS is certainly a computationally intensive task since it is predicated on repeated calls to the optimizer, and the overheads increase exponentially with ESS dimensionality. However, for canned queries, it may be feasible to carry out an offline enumeration; alternatively, when a multiplicity of hardware is available, the contour constructions can be carried out in parallel since they do not have any dependence on each other.

Fourth, while `PlanBouquet` can directly work off the API of existing query optimizers, `SpillBound` is *intrusive* since it requires changes in the core engine to support plan spilling and monitoring of operator selectivities. However, our experience with PostgreSQL is that these facilities can be incorporated relatively easily – the full implementation required only a few hundred lines of code.

At first glance, our move to half-space pruning algorithms (which is currently intrusive, since half-space pruning is achieved through spilling) may appear surprising given that hypograph-pruning based approaches are preferable from an implementation perspective due to their non-invasive nature. However, we show in Appendix A, dependency on ρ is an *organic* characteristic of the entire class of hypograph-pruning algorithms – we are therefore forced to consider half-space pruning approaches in our quest to be platform-independent, since achieving half-space pruning while being non-invasive is currently not known.

Finally, while PlanBouquet is dependent on the highly variable parameter ρ, it is possible that ρ itself is a weak function of D. Therefore, when D becomes really large, SpillBound's quadratic dependency on D may make its bounds weaker than those of PlanBouquet. However, our experience suggests that this transition does not happen for the D settings that are typically seen in current applications.

8 Related Work

Our work materially extends the PlanBouquet approach presented in [3], which is the first work to provide worst-case guarantees for query processing performance. As already highlighted, the primary new contribution is the provision of a structural bound with SpillBound, whereas PlanBouquet delivered a behavioral bound. Further, the performance characteristics of SpillBound are substantively superior to those of PlanBouquet, as illustrated in the experimental study.

A detailed comparison to the prior literature on selectivity estimation issues is provided in [3]. For completeness, we summarize the salient features here. The prior work can be classified into the following three categories:

Improving Estimation Accuracy: A comprehensive survey on the standard estimation techniques is available in [7]. Typically, histograms are used in current systems for storing statistical summaries of attribute value distributions, and their use is based on untenable assumptions such as Attribute Value Independence (AVI). Recently [16] took a step towards removing the independence assumption, but their work is restricted to handling two-dimensional histograms, and is inefficient for databases subject to frequent updates.

Bounding Estimation Error Impact: Techniques to minimize the adverse impact of errors in selectivity estimations are proposed in [11]. However, they do not address of recovering from large errors, which are quite common in practice, and also do not provide any guarantees.

Plan-switching Approaches: Plan-switching techniques have been considered for over two decades, and include influential systems such as POP [10] and Rio [1]. The key difference of SpillBound and PlanBouquet with regard to this prior work is the provision of performance guarantees. Further, they use the optimizer's plan choice as the starting point, and re-optimize at run-time if the estimates are found to be significantly in error. In contrast, SpillBound (and PlanBouquet) always start executing plans from the *origin* of the selectivity space, ensuring both repeatability of the query execution strategy as well as controlled switching overheads.

9 Conclusions and Future Work

We presented SpillBound, a query processing algorithm that delivers a worst-case performance guarantee dependent solely on the dimensionality of the selectivity space ($D^2 + 3D$). This substantive improvement over PlanBouquet is achieved through a potent pair of conceptual enhancements: half-space pruning of the ESS thanks to a spill-based execution model, and bounded number of executions for jumping from one contour to the next. The new approach facilitates porting of the bound across database platforms, easy knowledge of the bound value, low magnitudes of the bound, and indifference to the efficacy of the anorexic reduction heuristic. Further, our experimental evaluation on complex high-dimensional OLAP queries demonstrated that SpillBound provides competitive guarantees to its PlanBouquet counterpart, while the empirical performance is significantly superior.

In our future work, we intend to look into developing automated assistants for guiding users in deciding whether to use the native query optimizer or SpillBound for executing their queries. We also plan to work on extending SpillBound to handle the case of dependent predicate selectivities. Finally, we wish to leverage the empirical observation that plan cost functions are typically concave across the selectivity space, to further tighten the MSO guarantees of SpillBound.

References

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

[2] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for sql queries. In *ACM SIGMOD Conf.*, 2004.

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

[4] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 1993.

[5] D. Harish, P. Darera, and J. Haritsa. On the production of anorexic plan diagrams. In *VLDB Conf.*, 2007.

[6] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB Conf.*, 2002.

[7] Y. Ioannidis. The history of histograms (abridged). In *VLDB Conf.*, 2003.

[8] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Conf.*, 1998.

[9] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? In *VLDB Conf.*, 2016.

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

[11] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1), 2009.

[12] T. Neumann and C. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *BTW*, 2013.

[13] PostgreSQL. http://www.postgresql.org/docs/8.4/static/release.html.

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

[15] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB Conf.*, 2001.

[16] K. Tzoumas, A. Deshpande, and C. Jensen. Efficiently adapting graphical models for selectivity estimation. *VLDB Journal*, 22(1), 2013.

[17] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigumus, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *IEEE ICDE Conf.*, 2013.

Appendix

A PlanBouquet's MSO Dependency on ρ

In this section we prove lower bounds on the MSO for the class of *Hypograph-Pruning Algorithms* such as PlanBouquet. The input to the algorithms is as given in Section 2.4, i.e, they have access to the query Q, the D epps, and the knowledge of POSP plans in the ESS. Let us denote the set of POSP plans by $\mathcal{P}$. For a plan $P \in \mathcal{P}$, the algorithm can pre-compute and store the set of all the locations in the ESS for which P is the optimal plan, denoted by $S(P)$. Further, the algorithm may even pre-compute and store the $Cost(P, q)$ for all $q \in S(P)$.

The hypograph-pruning algorithms considered in this section are of the following type. They explore a sequence of (P, B) pairs where $P \in \mathcal{P}$ is executed with a execution cost budget of B. If the plan execution reaches completion within the budget B, then, the exact selectivity location q_a is discovered. Otherwise, the algorithm can only infer that $q_a \not\preceq q$ for all $q \in S(P)$ such that $Cost(P, q) \leq B$. Let us denote the ith pair explored by the algorithm as (P_i, B_i). We say that an hypograph-pruning algorithm is deterministic, if, the pair explored in the $(k+1)$st step is determined solely by the sequence of (P_i, B_i) pairs, $i = 1, \ldots, k$. The aggregate cost of the execution cost budgets of the sequence of pairs explored by the algorithm before termination is taken to be its overall execution cost. It is easy to see that the PlanBouquet falls into this class and considers only the (P_i, B_i) pairs on the doubling isocost contours.

We refer to the location in the ESS where all the epps have selectivity of 1 by $\mathbf{1}$ and the location where all the epps have selectivity 0 by $\mathbf{0}$. We denote the set of all deterministic hypograph-pruning algorithms by $\mathcal{H}P$ and denote an algorithm in this set by $\mathcal{A}$. In the rest of the section, we abuse the notation $\mathtt{Seq}_q$ introduced in Section 2.3 to instead refer to the sequence of (plan, budget) pairs explored by an algorithm when $q_a = q$. The following claim is true for any $\mathcal{A} \in \mathcal{H}P$.

Claim A.1 *Let q_1 and q_2 be two selectivity locations of* ESS. *Let the sequence of (P_i, B_i) pairs that any $\mathcal{A} \in \mathcal{H}P$ explores to discover q_1 and q_2 be denoted by $\mathtt{Seq}_{q_1}$ and $\mathtt{Seq}_{q_2}$ respectively. Then either $\mathtt{Seq}_{q_1}$ is a prefix of $\mathtt{Seq}_{q_2}$ or $\mathtt{Seq}_{q_2}$ is a prefix of $\mathtt{Seq}_{q_1}$.*

Proof 8 *Let $\mathtt{Seq}_{\mathbf{1}} = \{(P_1, B_1), \ldots, (P_k, B_k)\}$ be the sequence explored by $\mathcal{A}$ when $q_a = \mathbf{1}$. Let $\mathtt{Seq}_q$ be the sequence that $\mathcal{A}$ explores when $q_a = q \in \mathtt{ESS}$. Consider the first step of $\mathcal{A}$ in the two cases of $q_a = \mathbf{1}$ and $q_a = q$. Since the information it has is same in the beginning, both the sequences explore the same pair (P_1, B_1). Inductively, if both the sequences have not discovered $\mathbf{1}$ and q respectively at the end of ith step, then, the information they have at the end of ith step is exactly same and hence, their next steps are same. Further, at any point, if the sequence discovers $\mathbf{1}$, then, it is also guaranteed to discover q since $\mathbf{1} \succ q$ (due to PCM). This establishes that either both of them discovered via the same sequence or q is discovered via strict prefix of $\mathtt{Seq}_{\mathbf{1}}$. Therefore, both $\mathtt{Seq}_{q_1}$ and $\mathtt{Seq}_{q_2}$ are prefixes of $\mathtt{Seq}_{\mathbf{1}}$. So, one of them must be a prefix of the other.*

We would like to highlight that the above claim is indeed satisfied by PlanBouquet. The above claim implies that, for a given input, every deterministic hypograph-pruning algorithm is completely characterized by its sequence $\mathtt{Seq}_{\mathbf{1}}$.

For a given input ESS and corresponding $\mathcal{P}$ and the cost of POSP plans at all the locations, we let ρ denote the maximum number of plans of same cost.

Theorem A.1 *There exists an* ESS *and corresponding POSP profile $\mathcal{P}$ for which the MSO of any algorithm $\mathcal{A} \in \mathcal{H}P$ is at least ρ.*

Proof 9 *We consider the following* ESS *with associated $\mathcal{P}$ plans with special cost structures as follows:*

1. *It has a special isocost contour $\mathcal{IC}$ with the maximum number of POSP plans* $\texttt{PL} = \{P_1, P_2, \ldots, P_\rho\}$*. The cost $\mathcal{IC}$ is denoted by* CC*.*

2. *Consider any potential plan $P \notin \texttt{PL}$. For all locations q below $\mathcal{IC}$, $Cost(P, q)$ is at most 1; for all locations q, on or above $\mathcal{IC}$, $Cost(P, q)$ is at least ρ*CC*. We would like to clarify that, here, P_i is used to only indicate a POSP plan and not to be confused with the notation of (P_i, B_i) pair for denoting ith step of an algorithm.*

3. *For each of the plans $P_i \in \texttt{PL}$, $Cost(P_i, q) \geq \rho\texttt{CC}$ if $q \in \mathcal{IC}$. Further, for every location q on $\mathcal{IC}$ for which P_i is not the POSP plan, $Cost(P_i, q)$ is at least ρ*CC*. For a location q below $\mathcal{IC}$, $Cost(P_i, q)$ is at most 1.*

It can be checked that the above cost structure over the entire set of plans satisfies the PCM property. Suppose $\alpha < \rho$ is the MSO of $\mathcal{A}$ on the above ESS*. Since the plans P_i for $i = 1, \ldots, \rho$ are distinct, there are ρ distinct locations $q_1, q_2, \ldots, q_\rho$ on $\mathcal{IC}$, such that the POSP plan for q_i is P_i. From the* ESS *properties, if any location q_i is discovered using a plan P which is different from the POSP plan P_i, then its overall execution cost is at least ρ*CC*. Since the POSP cost at q_i is* CC*, the MSO is at least ρ, which contradicts the assumption that $\alpha < \rho$. Thus, to discover each q_i, $\mathcal{A}$ executes the corresponding POSP plan P_i with a budget* CC*. Thus this budget is insufficient for discovering any other q_j for $j \neq i$. Therefore, we can conclude that each of the locations q_i are discovered at different steps of* $\texttt{Seq}_{\mathcal{A}}$*. Let q_k be the location that is discovered last in the sequence* $\texttt{Seq}_{\mathcal{A}}$*, among the locations $\{q_1, \ldots, q_\rho\}$, i.e, the sequence for q_k contains the sequence for other q_is as prefix. But, to discover each of the q_is, a separate cost* CC *has to be incurred. Thus, the sequence for q_k has to incur a cost of at least ρ*CC *before discovering q_k. Since the POSP plan P_k has cost* CC*, $\mathcal{A}$ has an MSO of at least ρ, contradicting our assumption.*

B Execution Times for a TPC-DS Query

We conducted an experiment wherein query response times were explicitly measured at run-time for both native optimizer and SpillBound. For this we have chosen TPC-DS query 96 for which there are three error-prone predicates from the tables *store_sales* (denoted by SS), *time_dim* (denoted by TD), *household_demographics* (denoted by HD), and *store* (denoted by ST). The error-prone (join) predicates are SS ⋈ HD, SS ⋈ TD, and SS ⋈ ST. The actual selectivity location, q_a, is (10%, 2%,13%) and the optimal plan took 760 seconds to complete the query. However, the native optimizer incurred a sub-optimality of 19.5 for its estimated selectivities.

For the same query, Table 4 shows in detail the sequence of plan executions and the selectivity learnt by SpillBound at the end of every contour. The three columns SS−HD, SS−TD, and SS−ST correspond to the epps, SS ⋈ HD, SS ⋈ TD, and SS ⋈ ST, respectively. The selectivity information learnt in each contour is indicated by boldfaced font. The table also shows that for each execution, the plan employed, and the overheads accumulated so far. A plan P executed in spilling mode is indicated with a p (small cap).

The execution sequence comprises of partial execution of plans spanning eight contours and seven distinct plans. The execution sequence ends with the full execution of plan P_{17} by returning query results to the user. Note that no more than 3 plans are executed on any of the contours.

The total time incurred by the execution is 5903 seconds and the empirical MSO is 7.7 compared to 19.5 for native optimizer.

Contour no.	SS−HD (plan)	SS−TD (plan)	SS−ST (plan)	Time (sec)
1	**0.01** (p_4)	0	**0.008** (p_2)	23
2	**0.07** (p_6)	**0.004** (p_7)	**0.3** (p_2)	123
3	**0.1** (p_6)	**0.01** (p_7)	**1** (p_2)	411
4	**0.3** (p_6)	**0.05** (p_7)	**3** (p_2)	960
5	**0.7** (p_6)	**0.1** (p_{12})	3	1388
6	**1.5** (p_6)	**0.2** (p_{12})	3	1940
7	**3** (p_6)	**0.4** (p_{12})	**5** (p_2)	3593
8	**10** (p_{21})	**2** (p_{17})	**13** (P_{17})	5903

Table 4: `SpillBound` execution on TPC-DS Query 96