Platform-Independent Robust Query Processing

Srinivas Karthik, Jayant R. Haritsa, *Fellow, IEEE*, Sreyash Kenkre, Vinayaka Pandit, and Lohit Krishnan

Abstract—To address the classical selectivity estimation problem for OLAP queries in relational databases, a radically different approach called `PlanBouquet` was recently proposed in [1], wherein the estimation process is completely abandoned and replaced with a calibrated discovery mechanism. The beneficial outcome of this new construction is that provable guarantees on worst-case performance, measured as Maximum Sub-Optimality (*MSO*), are obtained thereby facilitating robust query processing. The `PlanBouquet` formulation suffers, however, from a systemic drawback—the MSO 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 first 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. It does so by incorporating plan termination and selectivity monitoring mechanisms in the database engine. 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 simply by query inspection. We go on to prove that `SpillBound` is within an $O(D)$ factor of the *best possible* deterministic selectivity discovery algorithm in its class. We next devise techniques to bridge this quadratic-to-linear MSO gap by introducing the notion of *contour alignment*, a characterization of the nature of plan structures along the *boundaries* of the selectivity space. Specifically, we propose a variant of `SpillBound`, called `AlignedBound`, which exploits the alignment property and provides a guarantee in the range $[\mathbf{2D+2}, \mathbf{D^2+3D}]$. Finally, a detailed empirical evaluation over the standard decision-support benchmarks indicates that: (i) `SpillBound` provides markedly superior performance w.r.t. MSO as compared to `PlanBouquet`, and (ii) `AlignedBound` provides additional benefits for query instances that are challenging for `SpillBound`, often coming close to the ideal of MSO linearity in D. From an absolute perspective, `AlignedBound` evaluates virtually all the benchmark queries considered in our study with MSO of around **10** or lesser. Therefore, in an overall sense, `SpillBound` and `AlignedBound` offer a substantive step forward in the long-standing quest for robust query processing.

Index Terms—Selectivity estimation, plan bouquets, 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 [3], [4]. 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 [5], 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 [1], 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 [1]—for instance, with Query 19 of the TPC-DS benchmark, it goes as high as a million![1] More importantly, worrisomely large sub-optimalities are *not rare*—for the same Q19, the sub-optimalities for as many as 40 percent of the locations in the selectivity space are higher than 1,000.

As explained in [1], most of the previous approaches to robust query processing (e.g., [3], [6], [7], [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 [1], which provides, for the first time, a provable MSO guarantee. 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) [9] over the selectivity

- *S. Karthik and J.R. Haritsa are with the Database Systems Lab, Indian Institute of Science, Bangalore, Karnataka 560012, India. E-mail: {srinivas, haritsa}@dsl.serc.iisc.ernet.in.*
- *S. Kenkre, V. Pandit, and L. Krishnan are with IBM Research, Bangalore, Karnataka 560045, India. E-mail: {srekenkr, pvinayak, lohit.namboodiri}@in.ibm.com.*

Manuscript received 7 Sept. 2016; revised 24 Dec. 2016; accepted 19 Jan. 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
Recommended for acceptance by W. Lehner, J. Gehrke, and K. Shim.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2664827

1. Assuming that estimation errors can range over the entire selectivity space.

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

Fig. 1. Example query (EQ).

space. The PlanBouquet technique guarantees $MSO \leq 4 * |PlanBouquet|$.[2]

1.1 PlanBouquet

We describe the working of PlanBouquet with the help of the example query EQ shown in Fig. 1, which enumerates orders for cheap parts costing less than 1,000. To process this query, current database engines typically estimate three selectivities, corresponding to the two join predicates $(part \bowtie lineitem)$ and $(orders \bowtie lineitem)$, and the filter predicate ($p\_retailprice < 1{,}000$). 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 Fig. 1).

1.1.1 *Example Execution*

Given the above query, PlanBouquet constructs a two-dimensional space, called as Error-prone Selectivity Space (*ESS*) corresponding to the epps, covering their entire selectivity range ($[0,1] * [0,1]$), as shown in Fig. 2a.

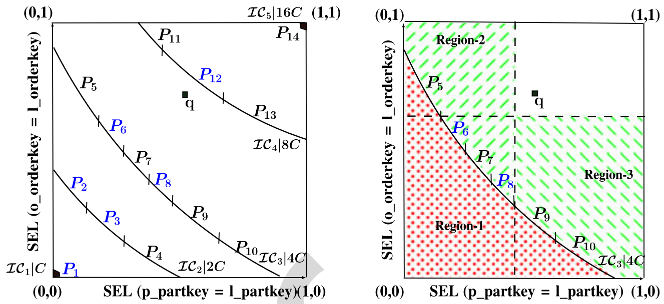
On this selectivity space, a series of *iso-cost* contours, $\mathcal{IC}_1$ through $\mathcal{IC}_m$, are drawn—each iso-cost contour $\mathcal{IC}_i$ has an associated cost $\mathtt{CC_i}$, and represents the connected selectivity curve along which the cost of the optimal plan, as determined by the 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.[3] That is, $\mathtt{CC_i} = 2^{(\mathtt{i}-1)}\mathtt{C}$ for $1 < i < m$. The last contour's cost, $\mathtt{CC_m}$, is capped to the maximum query cost at the top-right corner of the space.

As a case in point, in Fig. 2a, 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 Fig. 2a, 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*.

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, 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.

2. A more precise bound is given later in this section.
3. A doubling factor minimizes the MSO guarantee, as proved in [1].



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

Fig. 2. PlanBouquet and SpillBound.

As a sample instance, 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 Fig. 2a. 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.

1.1.2 *Performance Guarantees*

The overheads entailed by the "trial-and-error" exercise can be *bounded, irrespective of the query location in the space*. In particular, $MSO \leq 4 * \rho$, 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 Fig. 2a, the maximum density contour is $\mathcal{IC}_3$ which features six plans.

1.1.3 *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" [10] holding true.

1.2 SpillBound

Our objective here 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". Accordingly, we present a new query processing algorithm, called SpillBound, that achieves this objective in the sense that 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: (i) independent of the database platform,[4] (ii) known upfront by merely inspecting the query, and not incurring any preprocessing overhead, (iii) indifferent to the anorexic reduction heuristic, and (iv) certifiably low in value for practical values of D.

1.2.1 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 Fig. 2a, 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 percent over `PlanBouquet`.

The advantages offered by `SpillBound` are achieved by the following key properties—Half-space Pruning and Contour Density Independent (CDI) execution—of the algorithm.

1.2.2 Half-Space Pruning

With each contour whose plans do not complete within the assigned budget, `PlanBouquet` is able to prune the corresponding *hypograph*—that is, the search region *below* the contour curve.

A pictorial view is shown in Fig. 2b, which focuses on contour $\mathcal{IC}_3$—here, the hypograph of $\mathcal{IC}_3$ is the Region-1 marked with red dots.

However, with `SpillBound`, a much stronger *half-space*-based pruning comes into play. This is vividly highlighted in Fig. 2b, 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 in the execution plan tree are *prematurely terminated* at chosen locations, in conjunction with *run-time monitoring* of operator selectivities.

1.2.3 Contour Density Independent Execution

Let us define a "quantum progress" to be 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). Then, in the example scenario, while advancing through the various contours in the discovery process, `SpillBound` makes quantum progress by executing 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 quantum progress based on cost-budgeted execution of at most D carefully chosen plans on the contour.

Specifically, in each contour, for each dimension, one plan is chosen for spill-mode execution. 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 plans chosen for the contour $\mathcal{IC}_3$ along the X and Y dimensions, respectively.

4. Under the assumption that D remains constant across the platforms.

1.3 Bridging the MSO Gap

At this juncture, a natural question to ask is whether some alternative selectivity discovery algorithm, based on half-space pruning, can provide better MSO bounds than `SpillBound`. In this regard, we prove that *no* deterministic technique in this class can provide an MSO bound less than D. Therefore, the `SpillBound` guarantee is no worse than a factor $O(D)$ as compared to the best possible algorithm in its class.

1.3.1 Contour Alignment

Given this quadratic-to-linear gap on the MSO guarantee, we seek to characterize exploration scenarios in which `SpillBound`'s MSO approaches the lower bound. For this purpose, we introduce a new concept called *contour alignment*—a contour is aligned if the contour plan that is incident on the boundary of the ESS, has its selectivity learning dimension (during spill-mode execution) matching with the incident dimension. For instance, in the example of Fig. 2, contour $\mathcal{IC}_3$ would be aligned if plan P_5, rather than P_6, happened to be the plan providing the maximal guaranteed learning along the Y dimension. Leveraging this notion, we show that the MSO bound can be reduced to $O(D)$ if the contour alignment property is satisfied at *every contour* encountered during its execution.

Unfortunately, in practice, we may not always find the alignment property satisfied at all contours. Therefore, we design the `AlignedBound` algorithm which extracts the benefit of alignment wherever available, either natively or through an explicit induction. Specifically, `AlignedBound` delivers an MSO that is guaranteed to be in the platform-independent range $[\mathbf{2D+2}, \mathbf{D^2+3D}]$.

1.4 Empirical Results

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 six error-prone predicates, the MSO bound is 96 with `PlanBouquet`, but comes down to 54 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 from `PlanBouquet`'s 49 to 19 for `SpillBound`.

Turning our attention to `AlignedBound`, its performance is typically closer to the *lower end* of its guarantee range, i.e., $2D+2$, and often provides substantial benefits for query instances that are challenging for `SpillBound`. For instance, AlignedBound brings the MSO of the above-mentioned Q91 test case down to **10.4**. Moreover, AlignedBound is able to complete virtually all the benchmark queries evaluated in our study with a MSO of around 10 or lower.

In a nutshell, AlignedBound consistently collapses the enormous MSOs incurred with contemporary industrial-strength query optimizers, down to a single order of magnitude.

1.4.1 Caveats

We hasten to add that our proposed algorithms are *not* a substitute for a conventional query optimizer. Instead, they are 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, our algorithms are likely to be the preferred choice.

1.4.2 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 our algorithms are presented in Section 3.1. The SpillBound algorithm and the proof of its MSO bound are presented in Section 4, followed by the AlignedBound algorithm and its analysis in Section 5. The experimental framework and performance results are enumerated in Section 6, while pragmatic deployment aspects are discussed in Section 7. The related literature is reviewed in Section 8, and 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

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 $q_1, q_2 \in \texttt{ESS}$, 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.[5] 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 selectivity 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 [9].[6]

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 $\texttt{Seq}_{\texttt{q}_\texttt{a}}$.

2.3 Maximum Sub-Optimality [1]

We now present the performance metrics proposed in [1] 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

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

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

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

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 exploring a location $q \in \texttt{Seq}_{\texttt{q}_\texttt{a}}$—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:

$$SubOpt(\texttt{Seq}_{\texttt{q}_\texttt{a}}, \texttt{q}_\texttt{a}) = \frac{\sum_{\texttt{q} \in \texttt{Seq}_{\texttt{q}_\texttt{a}}} \texttt{Cost}(\texttt{P}_\texttt{q}, \texttt{q})}{\texttt{Cost}(\texttt{P}_{\texttt{q}_\texttt{a}}, \texttt{q}_\texttt{a})}, \quad (3)$$

5. For example, through the classical DP-based search of the plan space [11].

6. Letter subscripts for plans denote locations, whereas numeric subscripts denote identifiers.

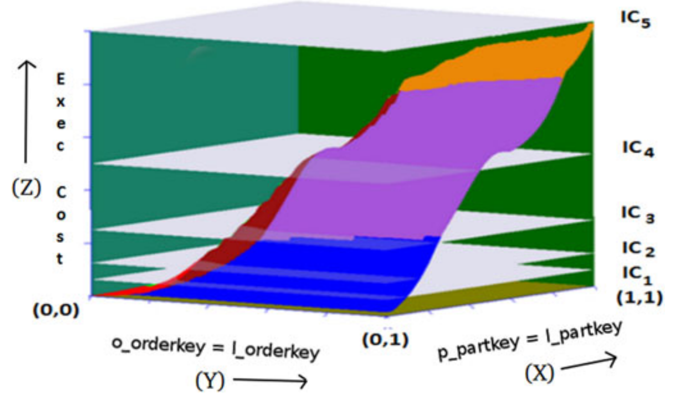


Fig. 3. 3D cost surface on ESS.

leading to

$$MSO = \max_{q_a \in \mathtt{ESS}} SubOpt(\mathtt{Seq}_{\mathtt{q_a}}, \mathtt{q_a}). \tag{4}$$

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* $q \in \mathtt{ESS}$*, develop a query processing approach that minimizes the MSO guarantee.*

As in [1], 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 $q_b, q_c \in \mathtt{ESS}$, and for any plan P,

$$q_b \succ q_c \Rightarrow Cost(P, q_b) > Cost(P, q_c). \tag{5}$$

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 extend 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 optimal plan on the ESS, i.e, for $q \in \mathtt{ESS}$, the value of the Z-axis is $Cost(P_q, q)$. This 3D surface, which captures the cost of the optimal plan 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 Fig. 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.[7]

7. Since Fig. 3 is only a perspective view of the OCS, it does not capture all the POSP plans.

TABLE 1
Notations

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 $q \in \mathtt{ESS}$
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 $\mathcal{IC}_i$

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 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 ith 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 ith 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 PLi. For example, in Fig. 3, PL4 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 $q \in \mathtt{ESS}$ 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.

3 Building Blocks of Our Algorithms

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. The AlignedBound algorithm that provides an $O(D)$ MSO under certain special scenarios is based on the concept of contour alignment. In this section, we present these building blocks of the SpillBound and AlignedBound algorithms.

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 using *spilling* during execution of query plans. While the use of spilling to accelerate selectivity discovery had been mooted in [1], 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

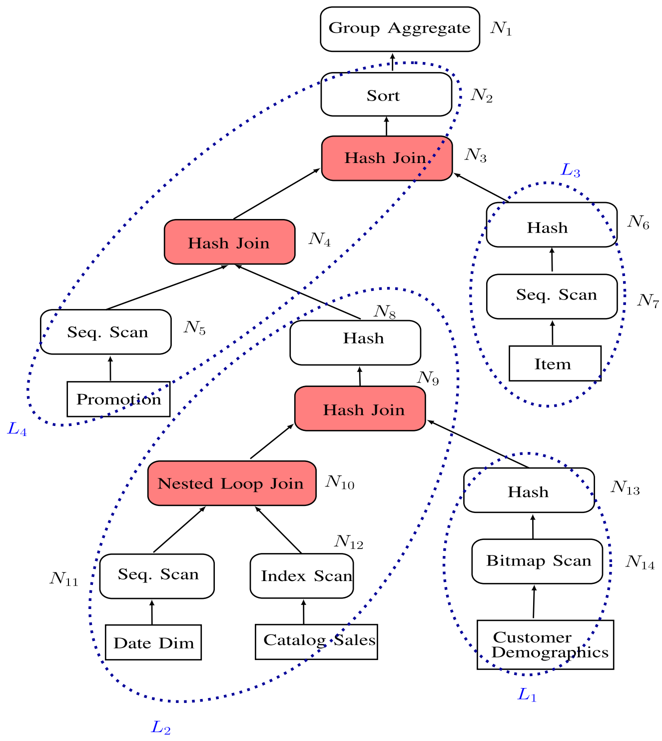


Fig. 4. Execution plan tree of TPC-DS Query 26.

plan tree [12]. 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* [13]. 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 Fig. 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 Spill-Mode of Execution

We now discuss how to execute plans in spill-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 of the complete plan tree as its *downstream* nodes.

Suppose we are interested in learning about the selectivity of an eppe_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 Fig. 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.

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 Fig. 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 Fig. 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" lemma. The proof of the lemma can be seen in [2].

Lemma 3.1. *Consider a plan P for which the spill node identification mechanism identifies the predicate e_j for spilling. Further, consider a location $q \in$ ESS. When the plan P is executed with a budget $Cost(P, q)$ in spill-mode, then we either learn (a) the exact selectivity of e_j, or (b) that $q_a.j > q.j$.*

3.2 Contour Density Independent Execution

We now show how the half-space pruning property can be exploited to achieve the contour density independent 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.

Consider the 2D ESS shown in Fig. 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_5^x, P_7^x, P_8^x in Fig. 5. The next step is to

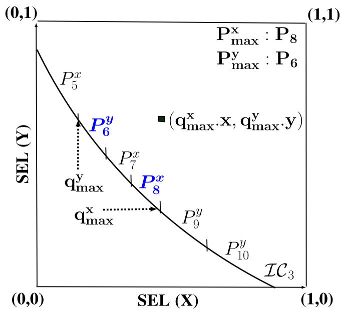


Fig. 5. Choice of contour crossing plans.

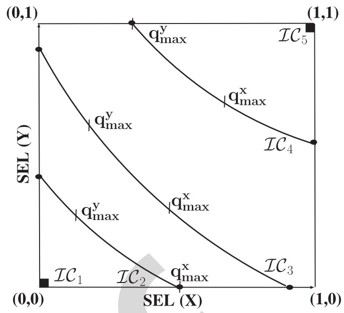


Fig. 6. Contour alignment.

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 Fig. 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 Fig. 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 $Cost(P_{q_a}, q_a) > \texttt{CC}_\texttt{i}$, triggering a jump to the next contour $\mathcal{IC}_{i+1}$.*

Proof. 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 $\texttt{CC}_\texttt{i}$. Thus $Cost(P_{q_a}, q_a) > \texttt{CC}_\texttt{i}$. □

Consider the general case of $\mathcal{IC}_i$ when there are more than two epps. Corresponding to an epp e_j, the location q^j_{max} and plan P^j_{max} are defined similar to the way q^x_{max} and P^x_{max} are defined (i.e, by replacing the X coordinate with the jth coordinate corresponding to e_j).

3.3 Contour Alignment

We now introduce a key concept that helps characterize search scenarios in which the MSO of the `SpillBound` algorithm matches the lower bound. Again, for ease of understanding, we consider the special case of a 2D ESS with predicates X and Y.

Consider a contour, say $\mathcal{IC}_i$, and a dimension $j \in \{X, Y\}$. A location $q^j_{ext} \in \mathcal{IC}_i$ is said to be an *extreme location along dimension* j if the location has the maximum coordinate value for dimension j among the contour locations belonging to $\mathcal{IC}_i$, i.e, $q^j_{ext}.j \geq q.j, \forall q \in \mathcal{IC}_i$. In Fig. 6, these extreme locations are highlighted by (bold) dots.

A contour $\mathcal{IC}_i$ is said to satisfy the property of contour alignment along a dimension j if it so happens that $q^j_{max} = q^j_{ext}$, i.e., the optimal plan at q^j_{ext} spills on predicate e_j. For ease of exposition, if a contour satisfies the contour alignment property along at least one of its dimensions, then we refer to it as an *aligned contour*. In Fig. 6, contours $\mathcal{IC}_2$ and $\mathcal{IC}_4$ are aligned along the X and Y dimensions, respectively, and are therefore aligned contours—however, contour $\mathcal{IC}_3$ is not so because it is not aligned along either dimension.

Given a contour $\mathcal{IC}_i$, Lemma 3.2 showed the sufficiency of *two* plan executions to guarantee a quantum progress in the discovery process. Leveraging the alignment notion, the following lemma describes when the same progress can be achieved with exactly *one* execution.

Lemma 3.3. *If a contour $\mathcal{IC}_i$ is aligned, then the execution of exactly one plan in spill-mode with budget $\texttt{CC}_\texttt{i}$, is sufficient to make quantum progress in the discovery process.*

Proof. Without loss of generality, let us assume that the contour $\mathcal{IC}_i$ satisfies contour alignment along dimension j, i.e, the optimal plan P at the location q^j_{ext} spills on dimension j. By Lemma 3.1, the spill-mode execution of P with budget $\texttt{CC}_\texttt{i}$ ensures that we either learn the exact selectivity of e_j or learn that $q_a.j > q^j_{ext}.j$. Suppose we learn that $q_a.j > q^j_{ext}.j$, then it implies that q_a lies beyond $\mathcal{IC}_i$. Thus, just the execution of P in spill-mode yields quantum progress. □

Note that in the general ESS case of more than two epps, there may be a *multiplicity* of q^j_{max} or q^j_{ext} locations, but Lemma 3.3 can be easily generalized such that quantum progress is achieved with a single execution in these scenarios also.

4 The 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}.

During the entire discovery process of `SpillBound`, only POSP plans on the isocost contour 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.

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 `2D-SpillBound`

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, as explained in [14].

4.1.1 Execution Trace

An illustration of the execution of 2D-SpillBound on TPC-DS Query 91 with two epps is shown in Fig. 7. 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 Fig. 7. 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.

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 results in the discovery of the actual selectivity of 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.

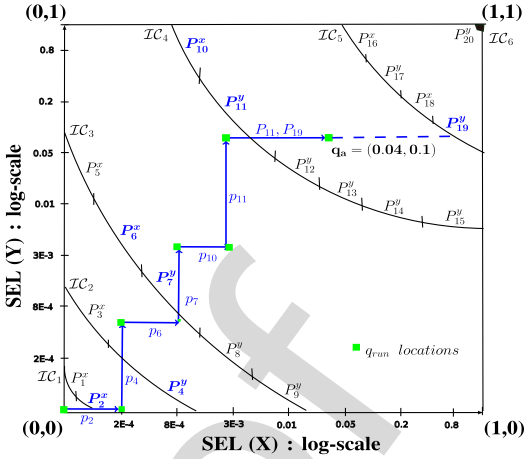


Fig. 7. Execution trace for TPC-DS Query 91.

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. Let the exact selectivity of one of the epps be learnt in contour $\mathcal{IC}_h$, where $1 \le h \le k+1$. From CDI execution, we know that 2D-SpillBound ensures that at most two plans are executed in each of the contours $\mathcal{IC}_1, \ldots, \mathcal{IC}_h$. 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:

$$\begin{aligned} TotalCost &\le 2 * \mathtt{CC_1} + \cdots + 2 * \mathtt{CC_k} + 3 * \mathtt{CC_{k+1}} \\ &= 2 * \mathtt{CC_1} + \cdots + 2 * 2^{k-1} * \mathtt{CC_1} + 3 * 2^k * \mathtt{CC_1} \\ &\le 2^{k+2} * \mathtt{CC_1} + 2^k * \mathtt{CC_1} \\ &= 5 * 2^k * \mathtt{CC_1}. \end{aligned} \tag{6}$$

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 $\mathtt{CC_k}$. By definition, $\mathtt{CC_k} = 2^{k-1} * \mathtt{CC_1}$. Hence,

$$MSO \le \frac{5 * 2^k * \mathtt{CC_1}}{2^{k-1} * \mathtt{CC_1}} = 10, \tag{7}$$

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 bound, $4 * 3 = 12$, 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. Further, when a contour $\mathcal{IC}_i$ is explored, the *effective search space* is the subset of locations on $\mathcal{IC}_i$ whose selectivity along the learnt dimensions matches the learnt selectivities. From now on, in the context of exploration, references to $\mathcal{IC}_i$ will mean its effective search space.

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 |EPP| plans that satisfy the half-space pruning property and ensure complete coverage of the contour. To do so, we consider the (location, plan) pairs $(q^1_{max}, P^1_{max}), \ldots, (q^{|\mathtt{EPP}|}_{max}, P^{|\mathtt{EPP}|}_{max})$ as defined at the end of the Section 3.2. The set of |EPP| plans that satisfy the contour density independent execution property is $\{P^1_{max}, \ldots, P^{|\mathtt{EPP}|}_{max}\}$.

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*.

Algorithm 1. The SpillBound Algorithm

```
Init: i = 1, EPP = {e_1, ..., e_D};
while i ≤ 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 IC_i, i.e, plans in PLi, and use this informa-
  tion 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
```

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($P^j_{max}, e_j, \mathtt{CC_i}$) refers to the execution of plan P^j_{max} spilling on e_j with budget $\mathtt{CC_i}$.

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

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

4.2.1 Performance Bounds

We now present an overview of how the MSO bound is obtained for SpillBound—the full proof is available in [14].

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 (proof in [14]):

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 $\frac{D(D-1)}{2}$.*

Suppose that the actual selectivity location 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 total cost incurred by the SpillBound algorithm in discovering q_a is 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 is incurred when all the repeat executions happen at the costliest contour, namely $\mathcal{IC}_{k+1}$. Hence, the total cost of SpillBound is given by

$$\sum_{i=1}^{k+1} (\#\text{fresh executions}(\mathcal{IC}_i)) * \mathtt{CC_i} + \frac{\mathtt{D(D-1)}}{2} * \mathtt{CC_{k+1}}.$$

Since the number of fresh executions on any contour is bounded by D, we obtain the following theorem (proof on similar lines to the 2D scenario):

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

Remark. For ease of exposition of SpillBound, and to facilitate comparison with PlanBouquet, we have chosen a cost ratio of 2 between successive contours. However, it is interesting to note that cost doubling is *not* the ideal choice for SpillBound, unlike PlanBouquet, as explained in [14]—for instance, a factor of 1.8 improves SpillBound's MSO guarantee from 10 to 9.9 in the 2D case. Only marginal improvements are obtained with these ideal factors for the ESS dimensionalities considered in our study.

4.3 Lower Bound

We now present a lower bound on MSO for a class of deterministic half-space pruning algorithms denoted by $\mathcal{E}$, that includes SpillBound in its ambit. We prove the following theorem.

Theorem 4.6. *For any algorithm $\mathcal{A} \in \mathcal{E}$ and $D \geq 2$, there exists a D-dimensional ESS where MSO of $\mathcal{A}$ is at least D.*

The proof of the above theorem is omitted due to space considerations and can be found in Section 5 of [14].

TABLE 2
Cost of Enforcing Contour Alignment

Query	Original	$\lambda = 1.2$	$\lambda = 1.5$	$\lambda = 2.0$	Max λ
3D_Q96	18	18	27	45	130
4D_Q7	70	70	90	90	3.62
4D_Q26	20	30	40	50	66.95
4D_Q91	67	67	77	77	5.38
5D_Q29	40	70	100	-	1.35
5D_Q84	100	-	-	-	1

5 The AlignedBound Algorithm

Given the quadratic-to-linear gap on MSO, we now identify exploration scenarios in which the MSO of `SpillBound` matches the $\Omega(D)$ lower bound—we do so by leveraging the contour alignment notion. Consider the scenario in which all the contours are aligned—then by Lemma 3.3, each of these contour requires only a single execution to make quantum progress. Following the lines of the analysis of `SpillBound`, and the fact that the most expensive execution sequence occurs when all the selectivities are learnt in the last contour ($\mathcal{IC}_{k+1}$), the total cost incurred in the worst-case would be

$$\begin{aligned} TotalCost &= \mathtt{CC_1} + \cdots + \mathtt{CC_k} + \mathtt{D} * \mathtt{CC_{k+1}} \\ &= \mathtt{CC_1} + \cdots + 2^{\mathtt{k}-1}\mathtt{CC_1} + \mathtt{D} * 2^{\mathtt{k}}\mathtt{CC_1} \\ &\leq (2^{k-1}\mathtt{CC_1})(2\mathtt{D} + 2), \end{aligned}$$

leading to the following theorem:

Theorem 5.1. *If the contour alignment property is satisfied at every step of the algorithm's execution, then the MSO bound is* $\mathbf{2D + 2}$.

In practice, however, the contour alignment property may not be natively satisfied at all contours—for instance, as enumerated later in Table 2, as few as 18 percent of the contours were aligned for a 3D ESS with TPC-DS Query 96. Therefore, we propose in this section the `AlignedBound` algorithm which operates in three steps: First, it exploits the property of alignment wherever available natively. Second, it attempts to *induce* this property, by replacing the optimal plan with an aligned substitute if the substitution does not overly degrade the performance. Finally, it investigates the possibility of leveraging alignment at a finer granularity than complete contours.

To aid in description of the algorithm, we denote by $Ext(i, j)$ the set of all extreme locations on a contour $\mathcal{IC}_i$ along a dimension j. With this, a contour $\mathcal{IC}_i$ is said to satisfy contour alignment along dimension j if $q^j_{max} \in Ext(i, j)$, i.e, at least one of the extreme locations along dimension j has an optimal plan that spills on e_j. Second, the set of all plans that spill on predicate e_k is denoted by $\mathcal{P}^k$.

5.1 Induced Contour Alignment

Given a contour $\mathcal{IC}_i$ that does not satisfy contour alignment, we *induce* contour alignment on the contour as follows: Consider a plan P which spills on $e_k \in$ EPP. It is a candidate replacement plan for any location $q^k_{ext} \in Ext(i, k)$ in order to obtain alignment along dimension k—the cost of the replacement is equal to $Cost(P, q^k_{ext})$. Therefore, the minimum cost of inducing contour alignment along dimension k is given by the pair $(P^k \in \mathcal{P}^k, q^k_{ext} \in Ext(i, k))$ for which $Cost(P^k, q^k_{ext})$ is minimized. Next, we find the dimension j for which the cost of the replacement pair (P^j, q^j_{ext}) is minimum across all dimensions. Finally, the optimal plan at q^j_{ext} is replaced by P^j, and the *penalty* λ of this replacement is the ratio of $Cost(P^j, q^j_{ext})$ to $Cost(P_{q^j_{ext}}, q^j_{ext})$.

The usefulness of induced contour alignment depends on the penalty incurred in enforcing the property. To assess this quantitatively, we conducted an empirical study, whose results are shown in Table 2. Here, each row is a query instance. The "Original" column indicates the percentage of the contours that satisfy contour alignment without any replacements. A column with a particular λ value, say c, indicates the percentage of the contours satisfying contour alignment when the replacement plans are not allowed to exceed a penalty of c. The last column shows the minimum penalty that needs to be incurred for all the contours to satisfy contour alignment.

We see from the table that there are cases where full contour alignment can be induced relatively cheaply—for instance, a 50 percent penalty threshold is sufficient to make Query 5D_Q29 completely aligned. However, there also are cases, such as 3D_Q96, where extremely high penalty needs to be paid to achieve contour alignment. Therefore, we now develop a weaker notion of alignment, called *"predicate set alignment" (PSA)*, which operates at a finer granularity than entire contours, and attempts to address these problematic scenarios.

5.2 Predicate Set Alignment

We say that a set $T \subseteq$ EPP satisfies predicate set alignment with the *leader dimension* j if, for any location $q \in \mathcal{IC}_i$ whose optimal plan spills on any dimension in T, $q.j \leq q^j_{max}.j$. The set of all locations in $\mathcal{IC}_i$ whose optimal plan spills on a dimension corresponding to a predicate in T, is denoted by $\mathcal{IC}_i|T$. For convenience, we assume that the predicate corresponding to the leader dimension belongs to T. Note that PSA is a weaker notion of alignment—while contour alignment with leader dimension j mandates that $q^j_{max}.j \geq q.j$ for any $q \in \mathcal{IC}_i$, PSA only requires that $q^j_{max}.j \geq q.j$ for all $q \in \mathcal{IC}_i|T$.

Lemma 5.2. *Suppose $T_1, \ldots, T_l$ are sets of epps satisfying predicate set alignment such that $\cup_{k=1}^{k=l} T_k =$ EPP, then $\cup_{k=1}^{k=l} \mathcal{IC}_i | T_k = \mathcal{IC}_i$.*

Proof. Every $q \in \mathcal{IC}_i$ spills on one of the dimensions in EPP. Therefore, it belongs to at least one $\mathcal{IC}_i|T$. □

Lemma 5.3. *Suppose $T_1, \ldots, T_l$ are sets of epps satisfying predicate set alignment such that $\cup_{k=1}^{k=l} T_k =$ EPP, then spill-mode execution of l POSP plans on $\mathcal{IC}_i$ is sufficient to make quantum progress.*

Proof. Let $j_1, \ldots, j_l$ be the leader dimensions for $T_1, \ldots, T_l$, respectively. Then, the l POSP plans chosen for the execution are $P_{q^{j_k}_{max}}$ for $k = 1, \ldots, l$. After this, based on the definition of PSA, Lemma 5.2, and an argument similar to the proof of Lemma 3.3, it can be shown that spill-mode execution of the chosen l POSP plans is sufficient to make quantum progress. The complete proof is available in [14]. □

5.2.1 Inducing Predicate Set Alignment

Consider a contour $\mathcal{IC}_i$, and a candidate set $T \subseteq$ EPP with a leader dimension $j \in T$. We now present a mechanism to induce predicate set alignment on T with leader dimension j.

We consider the extreme location along the dimension j among all the locations in $\mathcal{IC}_i|T$, i.e, $q_T^j = \arg\max_{q \in \mathcal{IC}_i|T} q.j$ (in case of a multiplicity of such points, any one point can be picked). Consider the set $S = \{q \in \mathcal{IC}_i \wedge q.j = q_T^j.j\}$, i.e, all the locations belonging to $\mathcal{IC}_i$ whose coordinate value on jth dimension is equal to the coordinate value on jth dimension of an extreme location in $\mathcal{IC}_i|T$. It is easy to see that T satisfies predicate set alignment if the optimal plan at any of the locations in S is replaced with a plan P that spills on e_j. We now find a pair $(P \in \mathcal{P}^j, q \in S)$ such that $Cost(P,q)$ is minimum. The predicate set alignment property is induced by replacing the optimal plan at q with the plan P. The penalty λ for the replacement is defined as before. We remark that the step of inducing predicate set alignment can be implemented efficiently and is discussed in [14].

5.2.2 Finding Minimum Cost Predicate Set Cover

Lemma 5.3 essentially says that a set of predicate sets $T_1, \ldots, T_l$ that cover EPP can be leveraged to make quantum progress. We now argue that it is sufficient to limit the search to merely the set of *partition covers* of EPP.

Consider a set T which satisfies PSA along dimension j. The *cover cost* of $T_1, \ldots, T_l$ is said to be sum of cost of enforcing PSA for each of the T_is. We say that T satisfies *maximal* PSA with leader dimension j if no super-set of T satisfies the property with same or lesser cost. Consider $T_1, \ldots, T_l$ which cover EPP and have been enforced to satisfy maximal PSA. We now obtain a partition cover whose cover cost is at most the cover cost of $T_1, \ldots, T_l$.

Let $j_1, \ldots, j_l$ be the leader dimensions for $T_1, \ldots, T_l$. The maximal property of the T_is implies that no dimension can be a leader dimension for more than one T_i. Therefore, the following sets $\pi_1 = T_1 + \{j_1\} - \cup_{m=2}^{m=l}\{j_m\}, \pi_k = T_k + \{j_k\} - \cup_{m=1, m\neq k}^{m=l}\{j_m\} - \cup_{m=1}^{m<k}\pi_m$ for $k = 2, \ldots l-1$, and $\pi_l = T_l - \cup_{m=1}^{m=l-1}\pi_m$ provide a partition cover with the same set of leader dimensions $j_1, \ldots, j_l$. It follows that the cover cost of $\pi_1, \ldots, \pi_l$ is at most the cover cost of $T_1, \ldots, T_l$. The full proof of this is presented in [14]. Therefore, we can restrict the search for EPP cover to only partition covers without incurring any increase in the penalty of the EPP cover. The benefit of this is that the number of partition covers of a set is much smaller than the number of different ways of covering a set with its subsets.

Given a partition cover $\pi = \{\pi_1, \ldots, \pi_l\}$, π_λ denotes the sum of the penalties incurred in enforcing PSA for each of the π_is along their leader dimensions.

5.3 Algorithm Description

The AlignedBound algorithm is presented in Algorithm 2. The steps that are identical to the steps in SpillBound are not presented again and simply captured as comments.

The key steps of the algorithm are **S**1 and **S**2 which are executed using the partition cover and predicate set alignment techniques described in Section 5.2.

A legitimate concern at this point is whether in trying to induce alignment, the $D^2 + 3D$ guarantee may have been lost along the way. The proof that this is not so, and that the quadratic bound is retained is available in [14]. In summary, AlignedBound delivers an MSO that is guaranteed to be in the platform-independent range $[\mathbf{2D+2}, \mathbf{D^2+3D}]$.

Algorithm 2. The AlignedBound Algorithm

```
Init: i = 1, EPP = {e_1, ..., e_D};
while i ≤ m do                                    ▷ for each contour
    /* Handle special 1-D case when it is encountered */
    S0: Π = Set of all partitions of EPP (remaining epps);
    S1: We pick π ∈ Π with minimum π_λ;
    for each part π_k ∈ π do
        S2: Let j_k be the leader dimension, P the replacement
            plan along dimension j_k, and q the location whose
            optimal plan is replaced with P;
        exec-complete = Spill-Mode-Execution(P, e_{j_k}, Cost(P, q));
        Update q_run.j_k based on selectivity learnt for e_{j_k};
        if exec-complete then
            Remove e_{j_k} from the part π_k and the set EPP;
            Break;
        end if
    end for
    /* Update ESS, jump contour as in SpillBound */
end while
```

6 Experimental Evaluation

As mentioned earlier, the MSO guarantees 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 MSO performance with that of PlanBouquet. Furthermore, we also conduct an evaluation of AlignedBound over the same set of queries to appraise its performance benefits over SpillBound. The experimental framework, which is similar to that used in [1], 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 benchmark, operating at the base size of 100 GB. 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.

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

The database engine used in our experiments is a modified version of PostgreSQL 8.4 [15] engine, with the primary changes being the (1) selectivity injection—to generate the ESS, (2) abstract plan execution—to instruct the execution engine to execute a particular plan, (3) time-limited execution of plans and (4) spilling—to execute plans in spill-mode. In addition, we implement a feature that obtains a least cost plan from optimizer which spills on a user-specified epp. This is primarily needed for AlignedBound algorithm to find the minimum penalty replacement pair which is mentioned in Section 5.

The remainder of this section is organized as follows. For ease of presentation, first we compare the performance of

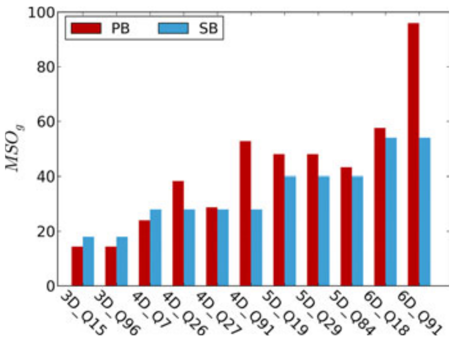


Fig. 8. Comparison of MSO guarantees (MSO_g).

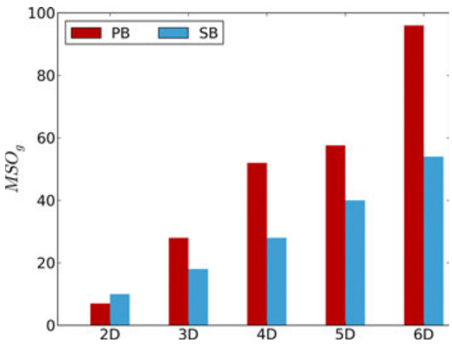


Fig. 9. Variation of MSO_g with dimensionality (Q91).

PlanBouquet and SpillBound, and subsequently move on to comparing SpillBound and AlignedBound. We use the abbreviations PB, SB and AB to refer to PlanBouquet, SpillBound and AlignedBound, 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 SpillBound versus PlanBouquet

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 [1], we conduct the experiments for PlanBouquet only after carrying out the *anorexic reduction* transformation [10] at the default $\lambda = 0.2$ replacement threshold—we use ρ_{RED} to refer to this reduced value.

6.2.1 Comparison of MSO Guarantees (MSO_g)

A summary comparison of MSO_g for PB and SB over almost a dozen TPC-DS queries of varying dimensionality is shown in Fig. 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_Q26, 4D_Q91 and 6D_Q91, SB's guarantee is noticeably *tighter* than that of PB—for instance, the values are 28 and 52.8, respectively, for 4D_Q91. In the remaining queries, the bound quality is roughly similar between the two algorithms. Therefore, contrary to our fears, the MSO guarantee is not found to have suffered due to incorporating platform independence.

6.2.2 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 corresponding performance profile is shown in

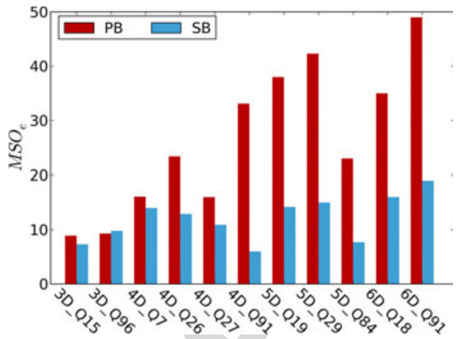


Fig. 10. Comparison of empirical MSO (MSO_e).

Fig. 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.

6.2.3 Comparison of Empirical MSO (MSO_e)

We now turn our attention to evaluating the empirical MSO, MSO_e, incurred by the two algorithms. There are two reasons that it is important to carry out this exercise: First, to evaluate the looseness of the guarantees. Second, 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 Fig. 10 for the entire suite of test queries. Our first observation is that the empirical performance of SB is far better than the corresponding guarantees in Fig. 8. In contrast, while PB also shows improvement, it is not as dramatic. For instance, considering 6D_Q18, PB reduces its MSO from 57.6 to 35.2, whereas SB goes down from 54 to just 16. A detailed analysis of the significant gap between SB's MSO_g and MSO_e values is provided in [2].

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_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 Fig. 10.

Finally, even for a query such as 4D_Q7, where PB had a marginally *better* bound (24 for PB and 28 for SB in Fig. 8), we find that it is SB which behaves better in practice (16.1 for PB and 13.9 for SB in Fig. 10).

6.2.4 Average-Case Performance (ASO)

A legitimate concern with our choice of MSO metric is that its improvements may have been purchased by degrading average-case behavior. To investigate this possibility, we have considered ASO, the average case equivalent of MSO, which is defined as follows under the assumption that all q_a's are equally likely

$$ASO = \frac{\sum_{q_a \in \text{ESS}} SubOpt(q_e, q_a)}{\sum_{q_a \in \text{ESS}} 1}. \quad (8)$$

We evaluated the ASO of PB and SB for all the test queries, and these results are shown in Fig. 11. Observe that, contrary to our fears, SB provides much better

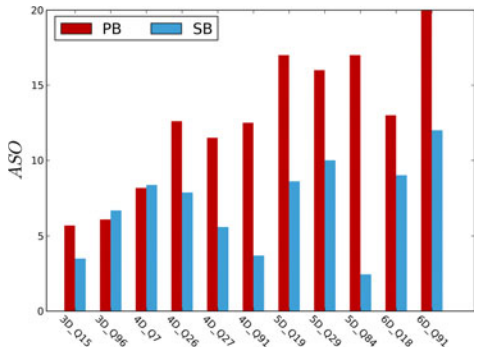


Fig. 11. Comparison of ASO performance.

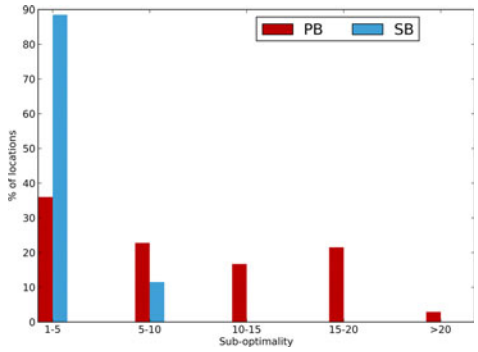


Fig. 12. Sub-optimality distribution (4D_Q91).

performance, especially at higher dimensions, as compared to PB. For instance, with 5D_Q19, the ASO for SB is nearly 100 percent better than PB, going down from 17 to 8.6. Thus, SB offers significant benefits over PB in terms of both worst-case and average-case behavior.

6.2.5 *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_Q91, is shown in Fig. 12, with sub-optimality ranges of width 5. We observe here that for over 90 percent of the ESS locations, the sub-optimality of SB is less than 5. Whereas this performance is achieved for only 35 percent 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.

6.3 Wall-Clock Time Experiments

All the experiments thus far were based on optimizer cost values. We have also carried out experiments wherein the actual *query response times* were explicitly measured for the native optimizer, SB and AB. As a representative example, we have chosen TPC-DS Q91 featuring 4 error-prone predicates, referred to as $e_1, \ldots, e_4$. In this experiment, the *optimal* plan took less than a minute (44 secs) to complete the query. However, the *native optimizer* required more than 10 minutes (628 secs) to process the data, thus incurring a sub-optimality of 14.3.

In contrast, SB took only around 4 minutes (246 secs), corresponding to a sub-optimality of 5.6. Table 3 shows the drilled down information of plan executions for every contour with SB. In addition, the selectivities learnt for the

TABLE 3
SpillBound Execution on TPC-DS Query 91

Contour no.	e_1 (plan)	e_2 (plan)	e_3 (plan)	e_4 (plan)	Time (sec.)
1	0	0	0	**0.08** (p_1)	1.3
2	**0.02** (p_3)	0	0	**0.3** (p_2)	7.5
3	**0.08** (p_4)	0	0	**1** (p_5)	21
4	**0.2** (p_4)	0	0	**12** (p_5)	51.2
5	**5** (p_9)	**0.8** (p_6)	0	12	86.3
5	**30** (p_9)	0.8	**5** (p_8)	**60** (p_7)	176.4
6	**80** (P_{10})	0.8	5	60	246.4

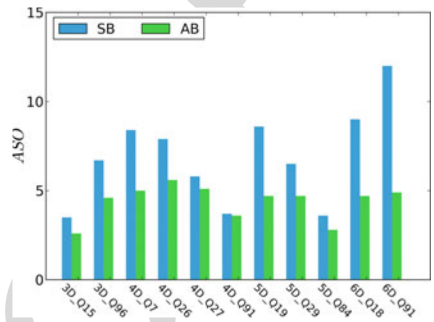


Fig. 13. Comparison of empirical MSO (MSO$_e$).

corresponding epp during every execution are also captured. The selectivity information learnt in each contour, shown in %, is indicated by boldfaced font in the table. Further, for each execution, the plan employed, and the overheads accumulated so far, are enumerated. A plan P executed in spill-mode is indicated with a p. As can be seen in the table, the execution sequence consists of partial executions of 13 plans spanning six consecutive contours, and culminates in the full execution of plan P_{10} which produces the query results.

Finally, we also conducted the above mentioned Q91 experiment with AB. The algorithm needed less than 3 minutes (165.1 secs) for completing the query, involving 10 partial plan executions before the culminating full execution. Thus, AB brings the sub-optimality down to just **3.8** in this example.

6.4 AlignedBound versus SpillBound

We now turn our attention to evaluating how the predicate set alignment property, exploited by AB, impacts its empirical performance as compared to SB. Specifically, we assess the MSO$_e$ incurred by the two algorithms, with the comparison on other metrics, such as ASO and sub-optimality distribution, deferred to [14].

6.4.1 *Comparison of Empirical MSO*

The MSO$_e$ numbers for SB and AB are captured in Fig. 13. First, we highlight that the MSO$_e$ values for AB are consistently less than around 10, for all the queries. Second, AB significantly brings down the MSO$_e$ numbers for the several queries whose MSO$_e$ values with SB are greater than 15. As a case in point, AB brings down the MSO$_e$ of 6D_Q91 from 19 to 10.4.

6.4.2 *Rationale for AB's Performance Benefits*

Recall that AB provides an MSO guarantee in the range $[2D+2, D^2+3D]$. As can be seen in Fig. 13, the MSO$_e$ values for AB are closer to the corresponding $2D+2$ bound value, shown with dotted lines in the figure. These results suggest that the empirical performance of AB approaches the $\mathcal{O}(D)$ lower bound on MSO.

We now shift our focus to examining the reasons for AB's MSO_e performance benefits over SB. In Table 4, the maximum penalty over all partitions encountered during execution is tabulated for the various queries. The important point to note here is that these penalty values are lower than 3, even for 6D queries. Since the highest cost investment for quantum progress in any contour is the maximum penalty times the cost of the contour, the low value for penalty results in the observed benefits, especially for higher dimensional queries.

TABLE 4
Maximum Penalty for AB

Query	max. penalty for AB
3D_Q15	2.42
3D_Q96	3
4D_Q7	2
4D_Q26	2.25
4D_Q27	2
4D_Q91	2.05
5D_Q19	2.5
5D_Q29	1.81
5D_Q84	1.1
6D_Q18	1.92
6D_Q91	1.25

6.5 Evaluation on the JOB Benchmark

All the above experiments were conducted on the TPC-DS benchmark, an industry standard. Recently a new benchmark, called Join Order Benchmark (JOB), specifically designed to provide challenging workloads for current optimizers, was proposed in [4]. Given its design objective, it appears appropriate to evaluate our query processing algorithms on this platform. A difficulty, however, is that all the queries in the JOB benchmark feature *cyclic predicates*, directly nullifying our selectivity independence assumption. Therefore, as an interim work-around, we shut off the optimizer's automatic inclusion of implicit join predicates, and verified that the consequent optimizer plans either remained the same or were only marginally sub-optimal.

We now present results for a representative Query 1a from the JOB benchmark. For this query, we found that, as expected by design, the native optimizer's performance was substantially worse, with the MSO going well above 6,000. In marked contrast, SB continued to retain its strong performance profile with an MSO of only around 12. And AB reduced this even further to below 9.

7 Deployment Aspects

Over the preceding sections, we have conducted a theoretical characterization and empirical evaluation of our proposed algorithms. We now discuss some pragmatic aspects of its usage in real-world contexts. Most of these issues have already been previously discussed in [1], in the context of the PlanBouquet algorithm, and we therefore only 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 $(1+\delta)^2$ [14]. That is, the MSO guarantee of SpillBound (and AlignedBound) would be $(D^2+3D)(1+\delta)^2$. Moreover, the errors induced by cost model are fairly small. For instance, $\delta = 0.3$ is reported in [16].

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 combination of 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.

Finally, while PlanBouquet can directly work off the API of existing query optimizers, SpillBound and AlignedBound are *intrusive* since they require 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.

8 Related Work

Our work materially extends the PlanBouquet approach presented in [1], 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 (and AlignedBound), whereas PlanBouquet delivered a behavioral bound. Further, the performance characteristics of both our algorithms 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 [1]. Since SpillBound and AlignedBound belong to the class of plan switching approaches, they may appear similar at first sight to influential systems such as POP [3] and Rio [6]. However, there are key differences: First, they start with the optimizers estimate as the initial seed and then conduct a full-scale re-optimization if the estimate is found to be significantly in error. In contrast, our proposed algorithms 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.

Second, both POP and Rio are based on heuristics and do not provide any performance bounds. In particular, POP may get stuck with a poor plan since its selectivity validity ranges are defined using structure-equivalent plans only. Similarly, Rios sampling-based heuristics for monitoring selectivities may not work well for join-selectivities, and its definition of plan robustness based solely on the performance at the corners of the ESS has not been validated.

9 Conclusion and Future Work

We presented SpillBound, a query processing algorithm that deliver 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 and low magnitudes of the bound value, and indifference to the efficacy of the anorexic reduction heuristic. Further, we also showed that SpillBound is within an $\mathcal{O}(D)$ factor of the best deterministic selectivity discovery algorithm in its class. Finally, we introduced the contour alignment and predicate set alignment properties and leveraged them to design AlignedBound with the objective of bridging the quadratic-to-linear MSO gap between SpillBound and the lower bound.

A detailed experimental evaluation on complex high-dimensional OLAP queries demonstrated that our algorithms provide competitive guarantees to their PlanBouquet counterpart, while their empirical performance is significantly superior. Moreover, AlignedBound's performance often approaches the ideal of MSO linearity in D.

In our future work, we wish to develop automated assistants for guiding users in deciding whether to use the native query optimizer or our algorithms for executing their queries. We also plan to work on extending SpillBound and AlignedBound to handle the case of dependent predicate selectivities.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and Anshuman Dutt for their valuable comments on this work. This paper is an extended version of [2], with the primary new contributions being the AlignedBound algorithm and its evaluation.

REFERENCES

[1] A. Dutt and J. Haritsa, "Plan bouquets: A fragrant approach to robust query processing," *ACM Trans. Database Syst.*, vol. 41, no. 2, pp. 11:1–11:37, 2016.
[2] S. Karthik, J. Haritsa, S. Kenkre, and V. Pandit, "Platform-independent robust query processing," in *Proc. 32nd IEEE Int. Conf. Data Eng.*, May 2016, pp. 325–336.
[3] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proc. ACM SIGMOD 30th Int. Conf. Manage. Data*, Jun. 2004, pp. 659–670.
[4] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" in *Proc. 42nd Int. Conf. Very Large Data Bases*, Sep. 2016.
[5] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO-DB2's learning optimizer," in *Proc. 27th Int. Conf. Very Large Data Bases*, Sep. 2001, pp. 19–28.
[6] S. Babu, P. Bizarro, and D. DeWitt, "Proactive re-optimization," in *Proc. ACM SIGMOD 31st Int. Conf. Manage. Data*, Jun. 2005, pp. 107–118.
[7] T. Neumann and C. Galindo-Legaria, "Taking the edge off cardinality estimation errors using incremental execution," in *Proc. 15th Conf. Database Syst. Business Technol. Web*, Mar. 2013, pp. 73–92.
[8] N. Kabra and D. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," in *Proc. ACM SIGMOD 24th Int. Conf. Manage. Data*, Jun. 1998, pp. 106–117.
[9] A. Hulgeri and S. Sudarshan, "Parametric query optimization for linear and piecewise linear cost functions," in *Proc. 28th Int. Conf. Very Large Data Bases*, Aug. 2002, pp. 167–178.
[10] D. Harish, P. Darera, and J. Haritsa, "On the production of anorexic plan diagrams," in *Proc. 33rd Int. Conf. Very Large Data Bases*, Sep. 2007, pp. 1081–1092.
[11] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proc. ACM SIGMOD 5th Int. Conf. Manage. Data*, Jun. 1979, pp. 23–34.
[12] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surveys*, vol. 25, no. 2, pp. 73–170, 1993.
[13] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating progress of execution for SQL queries," in *Proc. ACM SIGMOD 30th Int. Conf. Manage. Data*, Jun. 2004, pp. 803–814.
[14] S. Karthik, J. Haritsa, S. Kenkre, V. Pandit, and L. Krishnan, "Platform-independent robust query processing," Tech. Report TR-2016-02, DSL/CDS, IISc, 2016, http://dsl.cds.iisc.ac.in/publications/report/TR/TR-2016-02.pdf.
[15] PostgreSQL. [Online]. Available: http://www.postgresql.org/docs/8.4/static/release.html
[16] G. Lohman, "Is query optimization a solved problem?" [Online]. Available: http://wp.sigmod.org/?p=1075

Srinivas Karthik is currently working toward the PhD degree in the Indian Institute of Science, Bangalore. His interests include robust query processing and testing.

Jayant R. Haritsa is a faculty of the CDS and CSA Departments, Indian Institute of Science, Bangalore, India. His interests include database engine design and testing. He is a fellow of the IEEE and the ACM.

Sreyash Kenkre is a research staff member with IBM Research India, and is part of the Cognitive Industry Solutions Group. His interests include graph theory and algorithms.

Vinayaka Pandit is a senior technical staff member at IBM Research and leads the Cognitive Industry Solutions Group, IBM India Research Lab. His interests include approximation algorithms, combinatorial optimization and randomization techniques, and their application to information management problems.

Lohit Krishnan is a technical staff member at IBM Research and is part of the Cognitive Industry Solutions Group, IBM India Research Lab. His interests include database systems and machine learning.