Robust Query Processing

Srinivas Karthik *
srinivas@dsl.serc.iisc.ernet.in
Supervised by Jayant R. Haritsa *
*Database Systems Lab, Indian Institute of Science
Expected Graduation Date: July 2017

***Abstract*—*Robust query processing*, a long-standing problem, is about performance, predictability and ability to avoid sudden disruptions in performance. While query optimizer being an important component of query processing, my thesis focuses on query optimizer robustness which is mainly plagued with the problem of cardinality mis-estimates. Moreover, it is well known that these mis-estimates can lead to orders of magnitude sub-optimal performance.**

In order to mitigate this problem, there has been considerable work in the literature such as improving quality of statistical meta-data, feedback-based adjustments, on-the-fly re-optimization of queries, etc. Most of these previous approaches are based on heuristics and do not offer worst-case performance bounds. The notable exception to the heuristics based approaches is the `PlanBouquet` approach which provides a provable worst-case performance guarantees. In `PlanBouquet`, the selectivities are not estimated but are discovered at run-time using a set (or bouquet) of plans. The `PlanBouquet` formulation suffers, however, from a systemic drawback – the performance bound is a function of not only the query, but also the underlying database platform. Thus the bound value becomes highly variable. Moreover, it is infeasible to compute the bound value without substantial investments in preprocessing overheads.

We present `SpillBound`, *the first algorithm for robust query processing that provides a platform-independent worst-case bound*. 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. Overall, `SpillBound` offers a new platform-independent approach to robust query processing.

I. Introduction

In computer science, robustness can be defined as the ability of a computer system to cope with errors during execution. While in the context of data management, robustness is usually associated with recovery from failure, disaster preparedness, etc. In particular, *robust query processing* is about performance, predictability and ability to avoid sudden disruptions in performance. The importance of robustness in query processing is showcased by the fact that there have been two Dagstuhl seminars on this topic [3], [4]. Drilling down further, the notion of robustness in query processing has been categorized into the following three types [16]:

1) *query optimizer robustness:* "the ability of the optimizer to choose a good plan as expected conditions change"

2) *query execution robustness:* "the ability of the query execution engine to process a given plan efficiently under different runtime conditions"

3) *workload management robustness:* "characterizes how database system performance is vulnerable to unexpected query performance"

Although each of these robustness aspects has challenges, my thesis focuses on the query optimizer robustness.

Query optimizer robustness: Current optimizers use a cost model and a cardinality model for choosing a plan. Here cost provide an estimate of the time required for data processing which is a function of system hardware. On the other hand, cardinality indicates the quantity of data processing which is a function of data distribution and data correlations. It is well known that there are errors in cost model as well as cardinality estimates. However, Lohman, in [11], mentions that errors induced by cost model have limited impact (<30%, on an average). He adds to it saying that the same is not true with inaccurate cardinality estimates (of the intermediate results) which can lead to sub-optimal performance by orders of magnitude.

The reasons for error in cardinality estimation are due to coarse summaries, insufficient or stale system metadata, violation of simplifying assumptions and complex user-defined predicates [14]. Moreover, it has been shown that, even if estimation errors on the base relations are small the errors can propagate exponentially with respect to the number of joins in the query execution tree [9] leading to poor execution performance. There has been a lot of prior work in query processing in order to mitigate this problem which can be categorized into the following:

1) *Improving Accuracy:* A comprehensive survey on the standard estimation techniques is available in [8]. Typically, histograms are used in current systems storing the statistical summary of attribute values, and are based on assumptions such as Attribute Value Independence (AVI) and uniformity of data distributions. Recently [15] takes a step towards removing the independence assumption, but their work is restricted to handling two-dimensional histograms and is inefficient for databases subject to updates.

2) *Bounding Error Impact:* Techniques to minimize the adverse impact of errors in selectivity estimations are proposed in [6], [13]. However, [13] does not address the problem of recovering from large estimation errors. Moreover, both techniques run into the basic infeasibility of a single plan to be near-optimal across the entire selectivity space.

3) *Plan-switching Approaches:* Plan-switching techniques have been considered for over two decades, and include

influential systems such as POP [12] and Rio [1]. Further, they use optimizer's estimated plan as the initial seed, and then re-optimize if the estimates were found to be significantly in error. In particular, POP may get stuck with a poor plan since its validity ranges are defined using structure-equivalent plans only. Similarly, Rio's sampling-based heuristics for monitoring selectivities may not work well for join selectivities and its definition of plan robustness on the basis of performance at corners (principal diagonal) has not been justified.

Given this rich body of literature, none of them provide guarantees on the worst-case execution performance with the exception of a recently proposed approach called PlanBouquet [2].

A. *PlanBouquet*

In this approach, the compile-time estimation process is completely *abandoned* – instead, selectivities are *discovered* at run-time by observing the query completion status of a calibrated sequence of cost-limited executions from a carefully chosen set of plans, called the "plan bouquet". Conceptually, the plan bouquet can be viewed as a sequence of "isocost bucket"s, each consisting of a group of plans that are executed with a common cost budget, and the budget doubling from one bucket to the next. The buckets are explored sequentially, starting with the cheapest, and the process terminates when a plan is completely executed to completion within its assigned budget.

As mentioned before, a unique benefit of PlanBouquet is that it provides *guaranteed bounds* on worst-case execution performance. They use the notion of Maximum Sub-Optimality (**MSO**), introduced in [2], as a measure of the robustness of a query processing technique to errors in selectivity estimation. Specifically, given a declarative query MSO of a query processing algorithm is the worst-case ratio, over the entire selectivity space, of the cost expended by the algorithm with respect to the optimum cost incurred by an oracular system that magically knows the correct selectivities. Then, the MSO of PlanBouquet is bounded by $4 * \rho$, where ρ refers to the plan cardinality of the largest isocost bucket.

B. *Limitations of PlanBouquet*

The PlanBouquet formulation, while breaking new ground, suffers from a systemic drawback – the size of the plan bouquet, 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; (ii) it becomes infeasible to compute the value without substantial investments in preprocessing overheads. Moreover, ensuring a bound that is small enough to be of practical value, is contingent on the heuristic of "anorexic reduction" [5] holding true.

C. *SpillBound*

With an objective to develop a bound that is solely *query-dependent*, and not on the specifics of the platform on which the query is executing (i.e. a "structural bound" instead of a "behavioral bound"), we proposed a new query processing algorithm, called SpillBound in [10], that materially achieves this platform-independent objective. [1]

Specifically, SpillBound 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 $(D^2 + 3D)$.

SpillBound obtains its freedom from systemic dependencies through a potent pair of conceptual enhancements: First, it extends PlanBouquet's hypograph-based pruning of the selectivity discovery space to a much stronger halfspace-based pruning. Second, the calibrated advancement through the discovery space is attained with at most a fixed number, specifically D, of plan executions at each advance, whereas these advances may entail an arbitrary number of executions in PlanBouquet.

From a theoretical perspective, 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 formally showed that *no* deterministic algorithm based on half-space pruning can provide an MSO less than D. This result shows that the SpillBound bound is no worse than a factor $O(D)$ in comparison to the best possible algorithm in its class. The details of the lower bound proof can be seen in [10].

II. Problem Framework

In this section, we present the key concepts, notations, and the formal problem definition. Given a user query, current database engines typically estimate selectivities for the filter and join predicates. While it is conceivable that the filter selectivities may be estimated reliably, it is often difficult to ensure similarly reliable estimates for the join predicates. We refer to such predicates as error-prone predicates, or epp in short. For ease of presentation, we assume that the set of error-prone selectivity predicates for a given user query is known apriori.

A. *Error-prone Selectivity Space (ESS)*

Consider a query with D epps – the individual epps are denoted by $\{e_1, \ldots, e_D\}$, and the full set by 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 $q.j$.

[1]under the mild assumption that the number of predicates prone to estimation errors, is same across database platforms.

B. Search Space for Robust Query Processing

Suppose we have as input, a query and its epps. At query compilation time, 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 $q \in$ 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) [7].

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

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.

C. Maximum Sub-Optimality (MSO) [2]

Let us now precisely define the robustness metric MSO. Plan switching approaches like PlanBouquet and SpillBound 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 $\texttt{Seq}_{q_a}$. Specifically, suppose the discovery algorithm is currently 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. Thus the sub-optimality incurred by an algorithm which pursues $\texttt{Seq}_{q_a}$, relative to an oracle that magically knows q_a and therefore uses the ideal plan P_{q_a}, is defined as:

$$SubOpt(\texttt{Seq}_{q_a}, q_a) = \frac{\sum\limits_{q \in \texttt{Seq}_{q_a}} Cost(P_q, q)}{Cost(P_{q_a}, q_a)} \tag{1}$$

Moreover, the *maximum* sub-optimality that can potentially arise over the entire ESS is given by

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

D. Problem Definition

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

For a given input query Q along 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 MSO guarantee.*

The primary assumptions made in [2] and 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 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. This assumption is found to be true for virtually all the plans generated by PostgreSQL on the benchmark queries. On the other hand, SI assumes that the selectivities of the EPP 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.

The conceptual and material improvements offered by SpillBound are best understood in the wake of the limitations of PlanBouquet. Thus, we next present the working of PlanBouquet with the help of an example query EQ shown in Figure 1. In the subsequent Section, again with the assistance of EQ, we move on to describing SpillBound. EQ enumerates orders for cheap parts costing less than 1000. We assume the epps for EQ correspond to the two join predicates ($part \bowtie lineitem$) and ($lineitem \bowtie orders$), 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
```

Fig. 1: **Example Query (EQ)**

III. PlanBouquet [2]

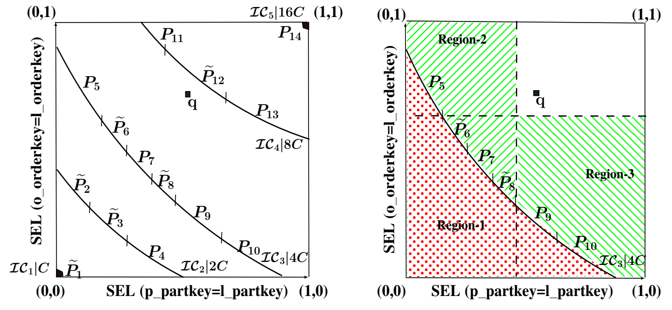
Example Execution: Given the above query, PlanBouquet constructs a two-dimensional space corresponding to the epps, covering their entire selectivity range $[0, 1]^2$, as shown in Figure 2(a). As mentioned before, every location in the space correspond to a particular combination of the selectivities of the epps. On this diagram, 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 $\texttt{CC}_\texttt{i}$, and represents the set of locations whose cost equal to $\texttt{CC}_\texttt{i}$. Further, the contours are selected such that the cost of the first contour $\mathcal{IC}_1$ corresponds to the minimum query cost C at the origin of the space, and the cost of each of the following contour is *double* that of the previous contour. That is, $\texttt{CC}_\texttt{i} = 2^{(i-1)}C$ for $1 < i < m$. The last contour's cost, $\texttt{CC}_\texttt{m}$, is capped to the maximum query execution cost at the top-right corner of the space.

In Figure 2(a), there are five hyperbolic-shaped contours, $\mathcal{IC}_1$ through $\mathcal{IC}_5$, with their costs going 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.* [2] If a plan fully completes its execution within the assigned time limit, then the results are returned to the user, and the algorithm terminates. 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, after which we move on to the next plan in the

[2]We assume a *perfect* cost model, although it can be relaxed to handle bounded cost model errors

contour and start all over again. In the event that all the plans in a contour have been tried out without any reaching completion, we move on to the next contour and the cycle repeats.



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

Fig. 2: PlanBouquet and SpillBound

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 will complete the query within the assigned budget. 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 region between contours $\mathcal{IC}_3$ and $\mathcal{IC}_4$, as shown in Figure 2(a). To process this query, PlanBouquet would invoke the 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 plan P_{12} completing the query.

Performance Guarantees and limitations: 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, they obtain $MSO \leq 4 * \rho$, where ρ is the plan cardinality on the "maximum density" contour, with density referring to the number of plans on a contour (in Figure 2(a), the maximum density contour is $\mathcal{IC}_3$ which features 6 plans). On the flip side, the specific value of ρ is *dependent* on both the query and the database operating environment. For instance, in the case of Query 25 of the TPC-DS benchmark with three epps, PlanBouquet's MSO guarantee which was 24 under PostgreSQL shot up, under the identical database and computing environment, to 36 for a commercial engine, due to the change in ρ.

IV. SpillBound [10]

While sharing the core discovery approach of PlanBouquet, the execution of SpillBound differs markedly. In the case of the example scenario in Figure 2(a), the sequence of budgeted executions would be (with associated plans having tilde symbol in Figure 2(a))

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

with P_{12} again completing the query – the reduced executions result in a cost savings of more than 50% over PlanBouquet. The advantages offered by SpillBound are achieved by the following two key properties of the algorithm.

1) 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 (rectangular region) that includes Region-2 (whole region which is to the left of vertical dotted line) is pruned by the (budget-limited) execution of P_8, while the half-space (rectangular region) that includes Region-3 (whole region which is below the horizontal dotted line) is pruned by the (budget-limited) execution of P_6. The 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. Specifically, this property is predicated on the following theorem.

Theorem 4.1: Consider a location $q \in$ ESS and the corresponding POSP plan P_q. Let e_j be a carefully selected epp such that there are no other epps in the subplan of P_q rooted at node corresponding to e_j. When the plan P_q is executed with budget $Cost(P_q, q)$ and spilling on e_j, then we either learn the exact selectivity of e_j, or infer that $q_a.j > q.j$.

2) 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 either cross a contour by executing (in spill-mode) at most D carefully chosen plans (one for each dimension) on the contour, or learn the exact selectivity of one of the epps (thus reducing the effective number of epps), irrespective of the actual number of plans on the contour. The plans are chosen in such a way that they provide 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.

V. Performance 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 value, we have carried out a detailed experimental evaluation of both approaches on standard TPC-H and TPC-DS benchmark queries, operating on the PostgreSQL engine.

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 drops from 52.8 with PlanBouquet to 28 with SpillBound. More pertinently, the *empirical* MSO of SpillBound is significantly better than that of PlanBouquet for *all* the queries. Here, empirical MSO refers to the maximum sub-optimality incurred by

SpillBound by exhaustively considering each and every location in the ESS to be q_a. For the same query Q91, the empirical MSO decreases drastically from PlanBouquet's 34 to just 7 for SpillBound. These empirical MSO numbers suggest the looseness of the MSO theoretical bound. More experimental results are captured in [10].

VI. Future Work

Now, let us see some of the interesting research directions for PlanBouquet based approach. The first four of these refer to the current limitations of our approach.

1) *Reducing preprocessing overheads:* 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. Furthermore, using multiple hardware, these overheads could be brought down significantly since the task is inherently parallelizable. Although it appears a feasible investment of time for canned queries, the same may not be true for ad hoc queries.

2) *Taxonomy on the usage of* SpillBound *and traditional optimizers:* the goal here is to provide a characterization of datasets or queries to decide on when to use SpillBound or traditional optimizers for a given user query. This potentially depends on the quality of selectivity estimates.

3) *Making* PlanBouquet *and* SpillBound *incremental with updates:* If the size of the database increases/decreases significantly then the already computed ESS would no longer be useful as of now. Thus a simple solution is to recompute the ESS again from scratch. It would be useful to develop an incremental solution with respect to updates.

4) *Relaxing the selectivity independence assumption:* In our work, we assume that the selectivities of the predicates in the query are independent with respect to each other. Often this assumption is *not* true in practice since the predicates could be correlated. Handling it in a naive way could lead to unreasonably high MSO bound values.

5) *Bridging the gap between $\mathcal{O}(D^2)$ upper bound and $\mathcal{O}(D)$ lower bound on MSO:* Currently, SpillBound offers an $D^2 + 3D$ ($\mathcal{O}(D^2)$) upper bound on MSO which is a factor $\mathcal{O}(D)$ away from the best possible algorithm in its space. We would like to design a new algorithm which reduces the upper bound further thereby moving closer towards the lower bound.

6) *Exploiting concavity in plan cost functions:* The previous objective of improving the MSO upper bound could be achieved by possibly exploiting the concavity property of plan cost functions on every projection. We have empirically observed, under PostgreSQL, that the majority of the plan cost functions obey this property across the selectivity space over standard benchmark query templates.

7) *Other performance metrics:* In this work, our objective has been to minimize the MSO which means that the worst-case sub-optimality incurred for any query instance in the ESS would be same. In certain practical scenarios, it would be desirable to relax the worst-case sub-optimality of some query instances for the more important ones. The importance for every query instance $q \in \texttt{ESS}$ could be modeled by assigning a relative weight $w(q)$. Thus the weighted version of MSO, denoted by WSO, can be defined as follows:

$$WSO = \frac{\sum_{q_a \in \texttt{ESS}} w(q_a) * SubOpt(\texttt{Seq}_{q_a}, q_a)}{\sum_{q_a \in \texttt{ESS}} w(q_a)} \qquad (3)$$

This weighted version of the problem has the following use cases: 1) by appropriately assigning weights one could achieve lesser worst-case sub-optimality for costlier queries than the cheaper ones (which is often desirable) 2) weights could also represent the likelihood of selectivities of a query instance coinciding with the actual selectivities 3) with all weights being equal refers to minimizing the average-case equivalent of MSO, referred to as ASO [2].

References

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

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

[3] G. Graefe, W. Guy, H. Kuno, and G. Paulley, editors. *Robust Query Processing, 5.08. - 10.08.2012*, Dagstuhl Seminar Proceedings 12321. Germany, 2012.

[4] G. Graefe, A. König, H. Kuno, V. Markl, and K. Sattler, editors. *Robust Query Processing, 19.09. - 24.09.2010*, Dagstuhl Seminar Proceedings 10381. Germany, 2010.

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

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

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

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

[9] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *ACM SIGMOD Conf.*, 1991.

[10] S. Karthik, J. Haritsa, S. Kenkre, and V. Pandit. Platform-independent robust query processing. Tech. Report http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2015-02.pdf.

[11] G. Lohman. Is query optimization a solved problem? http://wp.sigmod.org/?p=1075.

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

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

[14] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - db2's learning optimizer. In *VLDB Conf.*, 2001.

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

[16] J. Wiener, H. Kuno, and G. Graefe. Benchmarking query execution robustness. In *Performance Evaluation and Benchmarking*. Volume 5895 of LNCS Series, Springer, 2009.

Biography: Srinivas Karthik received master of technology degree in computer science and engineering from Indian Institute of Technology (IIT) Bombay, India in 2011. Then he joined IBM, India Research Lab and worked there for couple of years. From 2013, he is a Ph.D. student at Indian Institute of Science (IISc), working in the area of robust query processing.