Robust Query Processing

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFIMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Engineering
IN
Faculty of Engineering

BY

Kuntal Ghosh

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2016

Declaration of Originality

I, **Kuntal Ghosh**, with SR No. **04-04-00-10-41-14-1-11155** hereby declare that the material presented in the thesis titled

Robust Query Processing

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2014-2016**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discusions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: **Prof. Jayant R. Haritsa** Advisor Signature

DEDICATED TO

My Family and Friends

for their love and support

Acknowledgements

I am deeply grateful to Prof. Jayant R. Haritsa for his unmatched guidance, enthusiasm and supervision. He has always been a source of inspiration for me. I have been extremely lucky to work with him.

I am thankful to Anshuman Dutt and Srinivas Karthik for their assistance and guidance. It had been a great experience to work with them. My sincere thanks goes to my fellow lab mates as well for all the help and suggestions. Also I thank my CSA friends who made my stay at IISc pleasant, and for all the fun we had together.

Finally, I am indebted with gratitude to my parents and brother for their love and inspiration that no amount of thanks can suffice. This project would not have been possible without their constant support and motivation.

Abstract

In modern database systems, a query optimizer is used to estimate selectivities during plan selection for SQL queries. In practice, these estimates are often significantly different compared to the actual values encountered during query execution which results in highly sub-optimal execution performance. In [1], a different approach is proposed to address this classical problem, wherein the compile-time estimation process is completely skipped, rather query is executed through cost-limited executions of a small set of plans, called "**Plan Bouquet**". QUEST [2] is a prototype system, developed to showcase the concept of plan bouquet in existence. But, the earlier QUEST implementation was restricted to queries having at most two error-prone *base* predicates. In this work, we generalize the QUEST design and implementation so that any query having multidimensional error-prone predicates (*base* or *join*) can be evaluated.
For a given query, as we increase the number of error-prone predicates, the time taken to identify the parametric optimal set of plans (POSP) across the error-prone selectivity space (ESS) increases exponentially with the dimensionality of the space. In this work, we have improved the POSP identification time from *years* to *hours* using a massively parallel supercomputer, Cray XC40. We have also improved the performance (in terms of running time) of *basic CostGreedy* reduction algorithm [3] which is used to reduce the total number plans in POSP. Finally, extensive experiments were performed with a suite of multi-dimensional TPC-H based query templates on PostgreSQL optimizer. The results demonstrate that we can generate complex plan diagrams in significantly less time. Further, a complex plan diagram can be reduced substantially incurring only marginal increases in the estimated query processing costs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database systems accept declarative SQL queries as input. The queries should be executed efficiently so that data can be retrieved in less time. There are several alternative ways to execute a query which gives the same result. But, each alternate way of executing a query varies in performance.
Database systems have complex, cost based query optimizers which attempts to determine the most efficient way to execute a given query by considering all possible query plans. Query plan is an ordered sequence of steps to fetch the results for a SQL query. The optimizer will generate and evaluate many plans and choose the least cost plan as the best plan. After selection of a plan with minimum cost, it is passed to the executor module where the query is executed and results are shown as output.

1.1 Background

In database systems, accurate estimations of predicate selectivities are a basic requirement for the effective optimization of declarative SQL queries. In practice, however, these compile-time selectivity estimates are significantly different with respect to actual values encountered during query execution and that may result in highly sub-optimal plan selection.
In [1], a different approach is proposed to address this classical problem, wherein the compile-time estimation process is completely skipped, rather query is executed through cost-limited executions of a small set of plans, called "**Plan Bouquet**". This approach provides guarantees on worst-case performance for the first time in literature, but it requires a significant amount of pre-processing to be done to identify the bouquet of plans at compile-time.
The space formed by all possible combinations of erroneous selectivities is called as Error-prone Selectivity Space (ESS) [1]. Firstly, we identify the set of plans that covers the entire ESS. These

set of plans are called *Parametric Optimal Set of Plans(POSP)*. Secondly, a small bouquet of plans is identified from the POSP such that at least one of the bouquet plans is 2-optimal at each location in the space. In order to have practical guarantees, the POSP set is reduced using *CostGreedy* [3] plan reduction algorithm.

1.2 Motivation

QUEST [2] is a Java based prototype implementation of plan bouquet technique. It visually shows bouquet execution process and provides interactivity during execution. QUEST includes modified database system with incorporated features required for plan bouquet. Currently, PostgreSQL is the database system used with QUEST wherein all required features are implemented. At present, only two-dimensional ESS queries with error-prone *base* predicates can be evaluated through QUEST.

To evaluate a query with error-prone *join* predicates, we need *join-selectivity injection* feature to be implemented in a database so that we can get the optimizer estimated plan for a given join selectivity. But, open-source PostgreSQL doesn't have this feature implemented.

In order to make QUEST practically useful for queries having multi-dimensional error-prone predicates, we need to reduce the compile-time overhead of plan bouquet [1]. For a query, as we increase the error-prone dimensions, the time required to produce the POSP for the entire ESS increases exponentially with the dimensionality of the ESS. If the resolution of a d-dimensional ESS is res then, the number of optimization calls made for producing the POSP is res^d, which is computationally expensive.

1.3 Contribution

In this work, we generalize the QUEST design and implementation so that any multidimensional ESS query with error-prone *base* predicates as well as *join* predicates can be evaluated. We have also tried to reduce the time required to produce POSP for a given query template by using a massively parallel supercomputer, Cray XC40. Our contribution can be divided into three categories.

- **QUEST Modifications:** Here we explain new QUEST features including 'Query-coupled join selectivity injection' in PostgreSQL 9.4, query parser with error-prone predicates selection panel, a common database interface and query evaluation having multidimensional ESS.
- **A Massively Parallel Algorithm for POSP Identification:** Here, we propose a

parallel algorithm to reduce the time taken for POSP identification resulting in lesser pre-processing time for plan bouquet approach [1]. It also enables us to investigate high-dimensional plan diagrams [5] (a pictorial enumeration of the execution plan choices of a database query optimizer over the relational selectivity space) in high resolution.

- **Modified CostGreedy Algorithm:** In this chapter, we explain an optimization technique to speed-up *basic CostGreedy* algorithm.
  We'll also show in the experimental results that complex plan diagrams even with high resolutions can be reduced to a much simpler picture, featuring less number of plans without substantively affecting the query processing quality.

1.4 Organization

Chapter 2 provides the background details of plan bouquet technique. We explain the QUEST architecture in Chapter 3. Chapter 4 discusses the modifications in QUEST. In Chapter 5, we explain our parallel algorithm for POSP generation followed by a detailed description of *basic CostGreedy* algorithm modifications in Chapter 6. Chapter 7 represents the empirical evaluation of our approach. Finally, in Chapter 8, we summarize our work and mention the outline of future work.

Chapter 2

Plan Bouquet Technique

In this chapter, we present necessary background details about plan bouquet approach for robust query processing [1].

2.1 Overview

Plan bouquet is a new approach wherein the compile-time estimation process is completely shunned for error-prone selectivities. Instead, these selectivities are learnt systematically at run-time by executing a carefully chosen small set of plans called "Bouquet of plans" in a particular sequence in cost-limited manner. Partial executions are controlled by a graded progression of isocost surfaces projected onto the POSP curve. It has been proved in [1] that this construction results in guaranteed worst-case performance. It assumes Plan Cost Monotonicity (PCM) property, which states that cost of POSP plans increase monotonically with increasing selectivity values.

2.2 Single Dimension Example

Plan bouquet approach is explained for a simple query having one error-prone selectivity using query shown in Figure 2.1.

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

Figure 2.1: Example Query (EQ)

This example query contains one base predicate and two join predicates. Out of these,

assume selectivity of base predicate $p\_retailprice < 1000$ is error-prone.

2.2.1 Bouquet Identification

Firstly, the Parametric Optimal Set of Plans (POSP) that cover the entire selectivity range of the error-prone predicate is identified through repeated invocations of the optimizer and explicit injection of selectivities from lowest to highest values. Let POSP set be comprised of plans P1 through P5. Further, each plan is optimal over a certain selectivity range. The costs of these five plans, P1 through P5, over the entire selectivity range are enumerated as shown in Figure 2.2 (on a log-log scale). From these plots, the trajectory of the minimum cost from among the POSP plans is obtained as a curve which represents the ideal performance. This curve is called as POSP Infimum Curve (PIC). Next, the PIC is discretized by projecting a graded progression of isocost (IC) steps onto the curve. In Figure 2.2, the dotted horizontal lines represent a geometric progression of isocost steps, IC1 through IC7, with each step being double the preceding value. The intersection of each isocost step with the PIC (indicated by ■) gives an associated selectivity, and the identity of the best POSP plan for this selectivity.

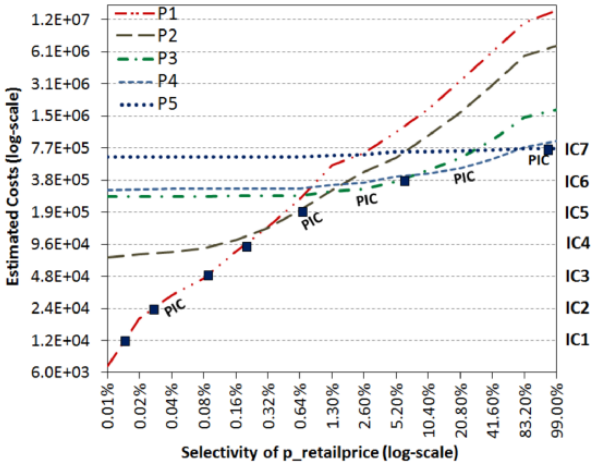


Figure 2.2: POSP performance

For example, in Figure 2.2, the intersection of IC5 with the PIC corresponds to a selectivity of 0.65% with associated POSP plan P2. The subset of POSP plans that are associated with the intersections forms the plan bouquet for the given query. So in Figure 2.2, the bouquet plans are P1, P2, P3, P5.

2.2.2 Bouquet Execution

At run-time, through a sequence of cost-limited executions of bouquet plans, the actual query selectivities are discovered. Specifically, execution begins with the cheapest isocost contour. Plans in each contour are sequentially executed with a budget limit equal to the contour's budget. If a plan completes its execution, then the results are returned to the user. Otherwise, the plan is forcibly terminated and next plan in the contour is selected for execution. If all the plans in a contour result in forcible termination, we move on to the next isocost contour and start all over again.

2.3 Extension to Multiple Dimensions

In multidimensional selectivity environment, the IC steps and the PIC curve become surfaces, and their intersections represent selectivity surfaces on which many bouquet plans may be present. In spite of these changes, the basic mechanics of the bouquet algorithm remain very similar to the 1D case. The primary difference is that we advance from one IC surface to the next only after it is determined that none of the bouquet plans present on the current IC surface can complete the execution of the given query within the associated cost budget.

2.4 Robustness Metric

Robustness is measured in terms of maximum sub-optimality **(MSO)** for plan bouquet.

Given a query Q and an ESS, let q_e denote the optimizers estimated query location, and q_a denote the actual run-time location in ESS. Also, denote the plan chosen by the optimizer at q_e by P_{oe} and the optimal plan at q_a by P_{oa}. Finally, let $cost(P_j, q_i)$ represents the execution cost incurred at actual location q_i by plan P_j. Then, robustness is defined by the normalized metric:

$$MSO = \max_{q_e, q_a \in ESS} [\frac{cost(P_{oe}, q_a)}{cost(P_{oa}, q_a)}]$$

Plan bouquet gives robustness guarantee in terms of MSO as:

$$MSO \leq 4\rho$$

If there exist K total contours and n_i plans lie on i^{th} contour then,

$$\rho = \max_{i=1 \text{ to } K} n_i$$

Chapter 3

QUEST Prototype System

QUEST [2] (QUery Execution without Selectivity esTimation) is prototype implementation of plan bouquet technique. It visually shows bouquet execution process and provides interactivity during execution. QUEST includes modified database system with incorporated features required for plan bouquet, and implementation of bouquet algorithms that execute query on modified database engine. Currently, PostgreSQL is the database system used wherein all required features are implemented.

3.1 System Architecture and Feature Details

The complete architecture of the QUEST system is shown in Figure 3.1, divided into a compile-time/pre-processing phase and a run-time/execution phase. In pre-processing phase, through repeated invocations of the optimizer, and explicit injection of selectivities, we identify a small bouquet of plans. In execution phase, a calibrated sequence of cost bounded executions of those plans are performed to complete the query execution.
QUEST features range from showing sub-optimality of native optimizer to executing query with bouquet mechanism. These features can be summarized as:

1. Performance comparison with native optimizer by showing native database execution and optimal execution performance.

2. Visually showing identification process of plan bouquet.

3. Actual and abstract executions of plan bouquet algorithm.

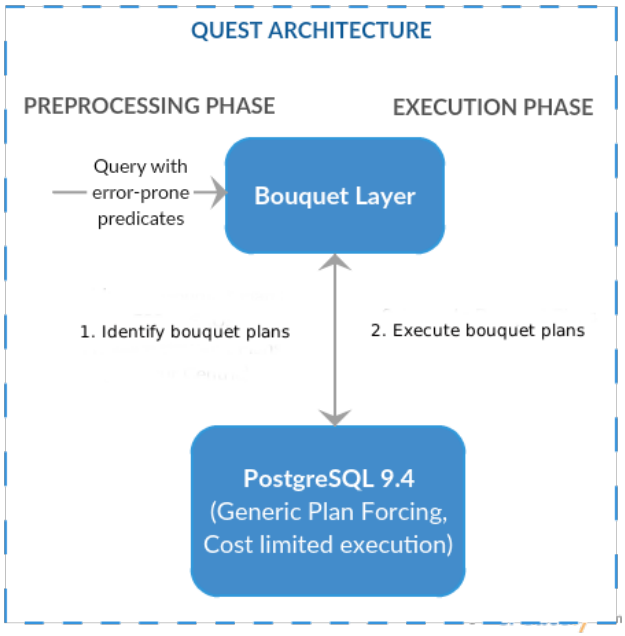


Figure 3.1: Existing QUEST Architecture

3.2 Implementation Details

QUEST interface is implemented using Java Swing. First, it validates the input query by accessing *pg_stats* meta-data relation. Next, through repeated invocations of the optimizer, and explicit *injection* of selectivities in PostgreSQL, we identify the POSP over the entire selectivity range (0-100%). *CostGreedy* reduction [3] is implemented for reducing POSP.
In this implementation, graphs and plan trees are drawn using open libraries "JFreeChart" [8] and "JGraph" [9] respectively. During bouquet execution, graphs are updated through functionality provided in JFreeChart. QUEST also provides functionality for clearing system cache. This cache clearing function runs system (Linux) commands through Java program.

Chapter 4

Modifications in QUEST

The new QUEST architecture is shown in Figure 4.1. In this chapter, we elaborate all the newly added QUEST features in the following order.

1. Query-coupled join selectivity injection in PostgreSQL 9.4.
2. Query parser with error-prone predicate selection panel.
3. A common database interface.
4. Query evaluation having multidimensional ESS.

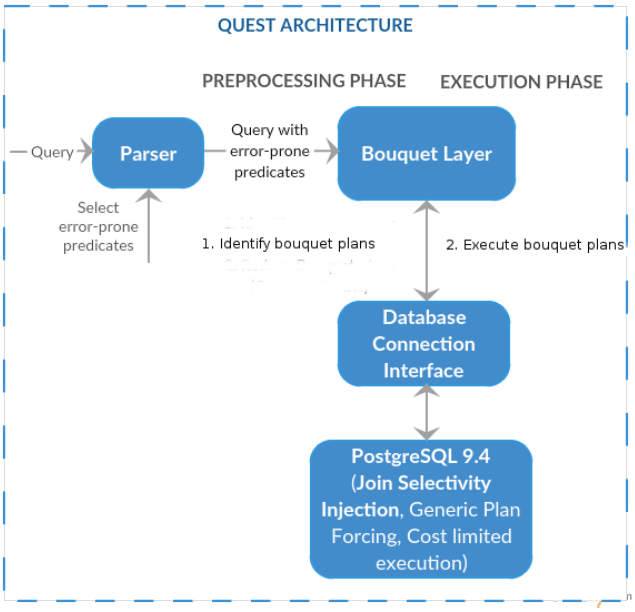


Figure 4.1: Modified QUEST Architecture

4.1 Query-coupled Join Selectivity Injection

Given a multidimensional query template and a selectivity space resolution, QUEST identifies the isocost contours (IC) in bouquet identification step. To achieve this objective, QUEST needs the optimizer estimated optimal plans and costs for different query locations in the ESS. Hence, we have implemented a feature in PostgreSQL to inject the user specified selectivity for inner-join predicates through the query and obtain the corresponding plan and cost from the optimizer. It can also be useful in some situations when we want to encourage some particular type of join operation or join order.

4.1.1 Implementation Details

In this section, we explain the implementation details of the join selectivity injection feature in PostgreSQL 9.4. To inject selectivities for join predicates, user has to use the following query format:

EXPLAIN SELECTIVITY (<*join_pred*1> AND <*join_pred*2> AND ...) (<*selec*1>,<*selec*2>,...) *statement*;

where, *selec*1 is the selectivity for *join_predicate*1, *selec*2 is the selectivity for *join_predicate*2 and so on. The injected selectivities should vary between 0 and 1.
We parse the input query and store injected join clauses and their corresponding selectivities in $PlannerInfo$ structure which stores all the information for planning a particular Query. PostgreSQL optimizer uses *clause_selectivity*() function to calculate selectivity of all kind of predicates. Whenever this function is called for the injected join predicates, we use the user-specified selectivity instead of the one calculated by the optimizer. Selectivities can only be injected for the join predicates which do not have any cyclic dependencies in the given query.

4.1.2 An Example

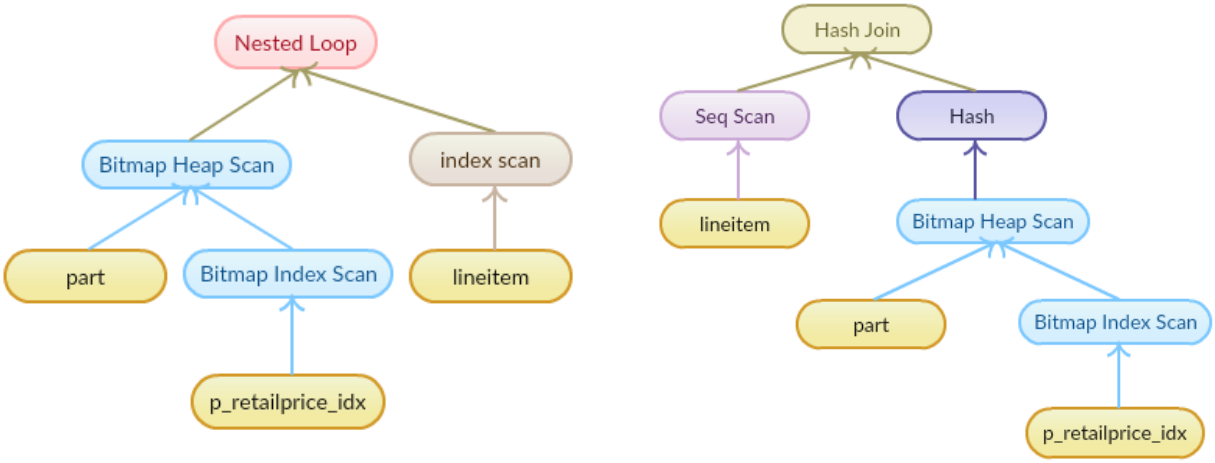
Let us compare the output of visual explain for the following query on TPC-H database with index created.

SELECT * FROM *lineitem*,*part* where *p_partkey* = *l_partkey* and *p_retailprice* <1000;

Here is the same query modified with different selectivity clauses:

EXPLAIN SELECTIVITY
($p\_partkey = l\_partkey$) (0.01) SELECT * FROM *lineitem*,*part* where $p\_partkey = l\_partkey$ and $p\_retailprice$ <1000;

EXPLAIN SELECTIVITY
($p\_partkey = l\_partkey$) (0.9) SELECT * FROM *lineitem*,*part* where $p\_partkey = l\_partkey$ and $p\_retailprice$ <1000;



(a) With selectivity of 0.01

(b) With selectivity of 0.9

Figure 4.2: Plan tree for different selectivity of $p\_partkey = l\_partkey$

Figure 4.2 shows the visual output of both the plan trees. Clearly, when the join predicate has lower selectivity, it is beneficial to use *Nested Loop Join* with index scan on the inner relation. Similarly, for higher selectivity of the join predicate, optimizer chooses *Hash Join*.

4.2 Query Parser with Error-prone Predicate Selection Panel

In QUEST, we have implemented a parser which parses the input query and extracts the base predicates and join predicates. Then, the user is asked to select the error-prone predicates among the extracted ones using a Java selection panel, see Figure 4.3.

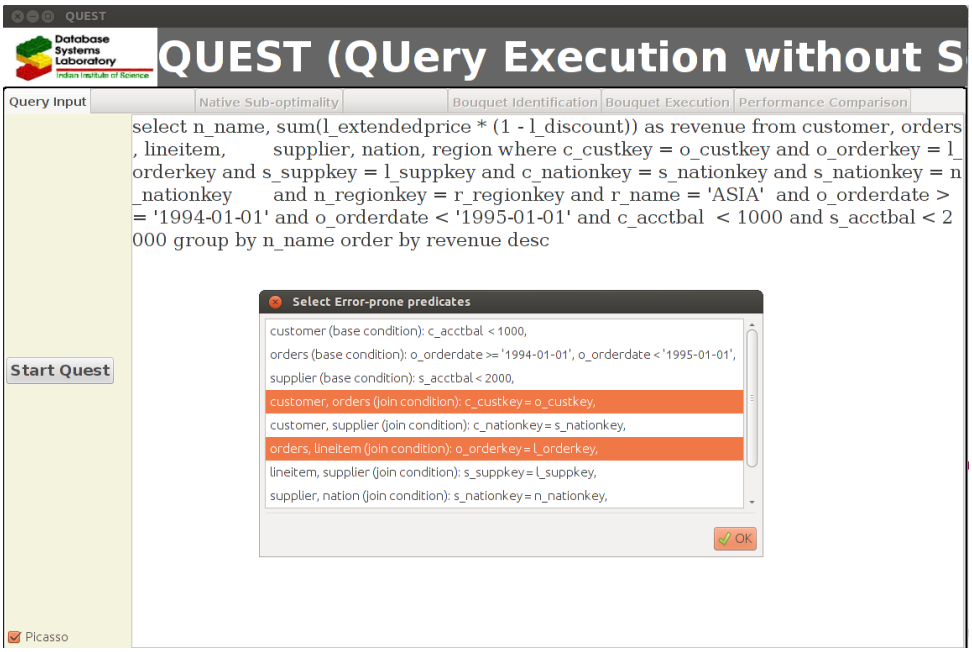


Figure 4.3: Error-prone Predicate Selection Panel

The implementation is similar to [6]. To achieve this, a regular expression based parser is implemented which collects the list of relations in the query and attribute names in the selected predicate in a tree structure. Tree structure is needed to handle nested queries correctly. The attribute name is searched in all relations present in the current scope to find the associated relation and schema.

4.3 A Common Database Interface

We have implemented a database interface module which abstracts away communication to the relational databases from QUEST. Different databases have slightly varying SQL syntax, different types of histograms and different plan representations. This module abstracts such inconsistencies under a common interface.

4.4 Query Evaluation having Multidimensional ESS

New QUEST implementation supports execution of query having multiple error-prone range predicates as well as join predicates. This is achieved by using an index array similar to [6]. QUEST lays out the plan number and cost information to single dimension for storage. The index array is used to calculate the position of each point in the single dimension layout. For now, we have made an assumption that the entire index array can be fitted in the available system memory.

Chapter 5

A Massively Parallel Algorithm for POSP Identification

For a given query, as we increase the number of error-prone predicates, the time taken to identify the parametric optimal set of plans (POSP) across the error-prone selectivity space (ESS) increases exponentially with the dimensionality of the space, assuming a fixed resolution.

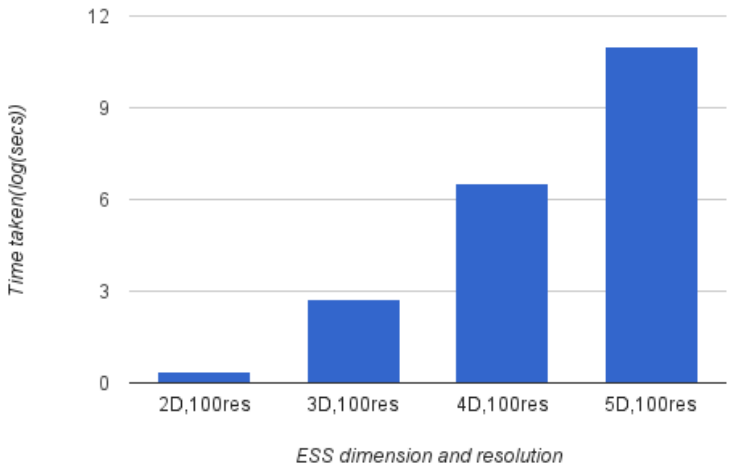


Figure 5.1: Time taken to generate plan diagrams

Figure 5.1 shows (on a log-scale) the estimated time taken on a system consisting of Intel Haswell Xeon processors with 128GB RAM to identify POSP by varying dimensions. To get a

feel about the identification time, it takes almost four years to generate POSP for a selectivity space with 5 dimensions and 100 resolution.
But, POSP identification time can be reduced significantly by efficient utilization of modern multi-core clusters. Here, we have developed an efficient parallel algorithm for POSP identification and implemented the same using *MPI*(Message Passing Interface) library.

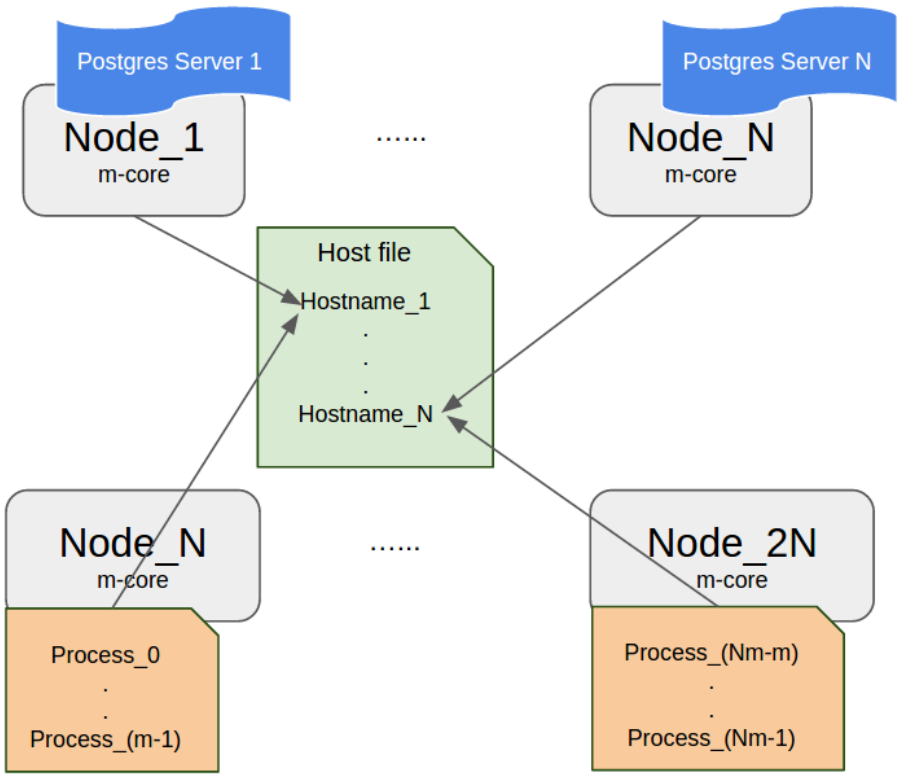


Figure 5.2: Parallel architecture for POSP Identification

Figure 5.2 depicts the system architecture of our implementation. Suppose, $2N$ (N is even) computing nodes are available and we want to use m cores in each node. Each process needs to connect to a database server. We use N nodes to start N database servers and rest of the nodes are used to execute the parallel algorithm. All the processes in i^{th} node connect to the database server hosted in $i/2^{th}$ node. A separate thread is created for each database connection. Since, we are creating m connections for each database server, all connection threads can be executed in a non-preemptive manner. Thus, the computational time for POSP identification is reduced by a factor of Nm.

Algorithm 1 Parallel Algorithm for POSP Identification

Input: number of processes n, dimension dim and resolution res of the ESS;

Output: POSP of the entire ESS;

1: Init: $curr_{qlocs}$ ={},$plans$ ={};
2: Call MPI_Init(); //Initialize MPI execution environment
3: Get $rank$ of the current process using MPI_Comm _rank();
4: **if** $rank = 0$ **then**
5: Divide the ESS ($dim * res$ points) into n equally sized subspaces;
6: Send n subspaces (defined by the top-right and bottom-left points) to n different processes using MPI_Send();
7: **end if**
8: Receive the subspace (S) information which needs to be explored by current process using MPI_Receive();
9: **for each** query locations $q \in S$ **do**
10: Get the optimal plan P_{opt} at q;
11: **if** $P_{opt} \notin plans$ **then**
12: Add P_{opt} in $plans$;
13: Add q in $curr_{qlocs}$;
14: **end if**
15: **end for**
16: Store $curr_{qlocs}$ in a shared buffer $qlocs$ using MPI_Gatherv();
17: $plans$ ={};
18: **if** $rank = 0$ **then**
19: **for each** query locations $q \in qlocs$ **do**
20: Get the optimal plan P_{opt} at q;
21: **if** $P_{opt} \notin plans$ **then**
22: Add P_{opt} in $plans$;
23: Store P_{opt} in disk;
24: **end if**
25: **end for**
26: **end if**
27: Call MPI_Finalize(); //Terminates MPI execution environment

The main idea of the our algorithm 1 is to divide the multi-dimensional ESS equally among the processes. Each process identifies POSP in its subspace. For each plan in POSP, a process

stores the query location where it was found for the first time. Finally, we find the POSP for the entire ESS by exploring only those query locations.

Along with POSP, we also store the cost and optimal plan at each location of the selectivity space in a linear data structure. This information can be used by QUEST during bouquet identification phase for a *parameterized* query template to avoid numerous optimizer calls. Hence, pre-processing time of Plan Bouquet can be substantially reduced.

Chapter 6

Modified CostGreedy Algorithm

For queries having multi-dimensional error-prone predicates, the worst-case bound of plan bouquet algorithm depends on ρ, the maximum plan density over the isocost surfaces. Therefore, to have a practically useful bound, we need to ensure that the value of ρ is kept to the minimum. This can be achieved through the *basic CostGreedy* algorithm, a plan reduction technique described in [3]. Here, POSP plans are allowed to swallow other plans, that is, occupy their regions in the ESS space, if the suboptimality introduced due to these swallowings can be bounded to a user-defined threshold, λ.
The input to the algorithm is a plan diagram [5], a pictorial enumeration of the execution plan choices of a database query optimizer over the relational selectivity space and a threshold λ. The output is another plan diagram. The data structures used in the algorithm are as follows:

1. $cur(q)$: current plan of a point q in the plan diagram.
2. $belong(q)$: the set of plans that can be used instead of the current plan in the reduced plan diagram.
3. $cost(q)$: value indicating the cost of q in the plan diagram.
4. $color(q)$: integer denoting the color (equivalently, plan) of q in the plan diagram.

The time complexity of CostGreedy is $O(mn)$, where m and n are the number of query points and plans, respectively, in the input plan diagram P [3].

Step 3 of *CostGreedy* algorithm (2) updates $belong(q)$ with all the plans that are in q's first quadrant with cost within the given threshold starting from *TopRight* to *BottomLeft*. But, if *plan-cost monotonicity* (PCM) assumption holds then, we don't need to check all the points in

q's first quadrant to find plans having cost within the threshold.

Algorithm 2 Basic CostGreedy Algorithm [3]

Input: Plan Diagram P, Threshold λ;

Output: Reduced plan diagram;

1: **for each** query locations q from *TopRight* to *BottomLeft* **do**
2: set $cur(q) = color(q)$;
3: update $belong(q)$ with plans that are in q's first quadrant with cost within the given threshold.
4: **end for**
5: Let m be the number of points in P and n be the cardinality of POSP for P.
6: Create n sets $S = \{S1, S2, \ldots Sn\}$ corresponding to the n plans.
7: Let $U = \{1, 2, \ldots m\}$ correspond to the m query points.
8: Define $\forall i = 1, \ldots, n, S_i = \{j : i \in belong(r)$ or $i = cur(r)$ for query point r corresponding to $j, \forall j = 1, \ldots, m\}$.
9: Let $I = (U, S)$, I be an instance of the Set Cover problem.
10: Let L_n be the color of the $TopRight$ point. Remove set S_n and all its elements from I.
11: Apply Algorithm Greedy Setcover to I. Let C be the solution.
12: $C = C \bigcup S_n$
13: Recolor the grid with colors corresponding to the sets in C and update new costs appropriately. If a point belongs to more than one subset, use color that results in least cost increase.
14: End Algorithm CostGreedy.

We need to modify the $belong(q)$ data structure so that it also includes the current plan at q. We use $belong(q)$ at step 8 of the algorithm which can be changed accordingly, because, we are already making sure that $belong(q)$ includes the current plan at q.

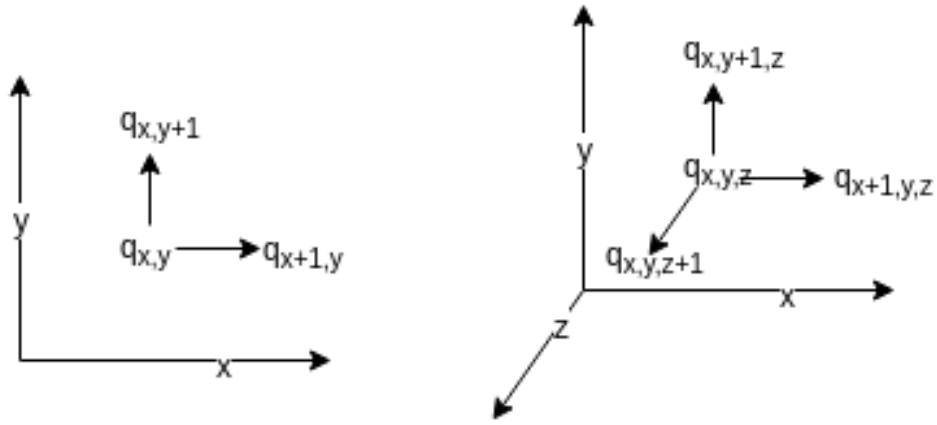


Figure 6.1: 2D and 3D Example

Now, suppose that $q_{x,y}$ is a query location in 2D ESS (Figure 6.1). Then, we only need to check plans only at $q_{x+1,y}$ and $q_{x,y+1}$ to find $belong(q_{x,y})$. Similarly, for 3D ESS, we need to check plans at $q_{x+1,y,z}$,$q_{x,y+1,z}$ and $q_{x,y,z+1}$ to calculate $belong(q_{x,y,z})$. Hence, for a d-dimensional ESS we need to check plans at d locations to find $belong(q)$ for a query location q.

Claim: *If PCM assumption holds, then,* $belong(q) \subseteq \bigcup_{i=1}^{d} belong(q_i) \cup p$, *where* q_i*s are the immediate neighbour points of* q *(at* q*'s first quadrant) along each of the* d *dimensions and* p *is optimal plan at* q.

Proof: Suppose, the cost of the plan p at location q is c. For another location q', the cost corresponding to the optimizer chosen plan p' is c'. Suppose, $p' \in belong(q)$ which means $c \leq c' \leq c(1+\lambda)$ where λ is the user-defined threshold. Due to PCM assumption, q' must be in q's first quadrant. It is easy to see that p' can be used for any point in the region bounded by q and q'. Hence, for any plan other than p that can be used at q, there is at least one point among q's neighbours (in q's first quadrant) where it can complete execution with cost within the threshold. We need to check all neighbour points along each dimension since PCM assumption doesn't provide any relation between these points. ■

After the above modifications in the *CostGreedy* algorithm, although asymptotically the time complexity is still $O(mn)$, but empirically we get significant performance improvement. Because, for most of the query points, $belong(q)$ contains $1\% - 5\%$ of the total number of plans in the ESS. We'll show the results in the next chapter.

Chapter 7

Empirical Analysis

In this chapter, we report an experimental evaluation of our proposed techniques.

7.1 Experimental Environment

PostgreSQL version 9.4 is used to obtain estimated optimal plans and estimated cost of plans. Our experiments have been carried out on Cray XC40 composed of 1376 computing nodes [12]. Each node has 2 CPU sockets with 12 cores each, 128GB RAM and they are connected using Cray Aries interconnect. Concerning software environment, the entire system is built to operate using Cray's customized Linux OS, called Cray Linux Environment. For our parallel algorithm implementation, we have used the GNU compiler collection with Cray MPICH2 Message Passing Interface and for *CostGreedy* reduction algorithm, we have used Java. We have performed all the experiments on TPC-H 1GB database [10].

7.2 Impact of Parallel Algorithm

We have used 300 nodes and 20 cores in each node for our experimental purpose. As discussed in Chapter 5, we get 3000 times speed-up. Figure 7.1 indicates the performance improvement of parallel algorithm over serial one.

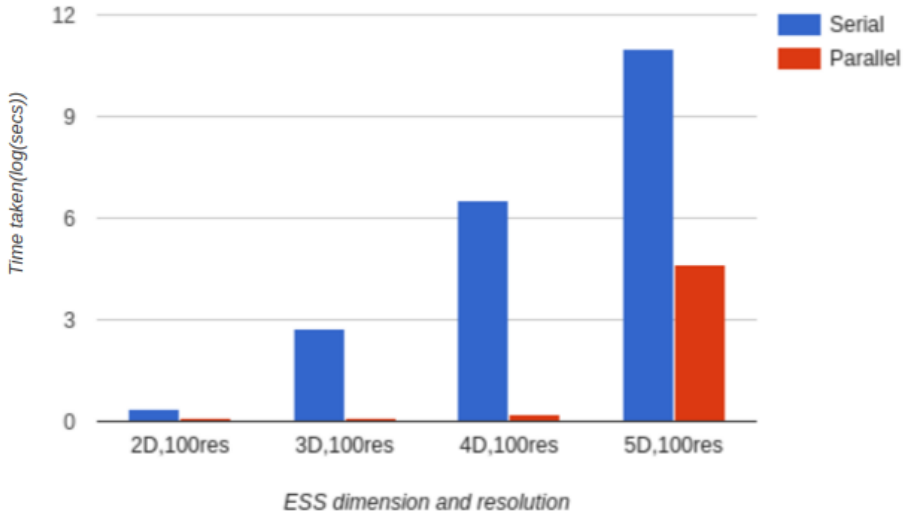


Figure 7.1: POSP Identification performance

As we can see in the figure, for a query with 5-dimensions and 100 resolution, our algorithm reduces the running time from *4 years* to *12 hours*. This enables us to generate POSP as well as the complete plan diagrams for high-dimensional ESS with high-resolutions in very less time which was previously unthinkable. It also reduces the compile-time overhead for plan bouquet to a great extent. For *parameterized* query templates, we can reuse these plan diagrams for bouquet identification phase in QUEST.

7.3 Modified CostGreedy Algorithm Performance

Figure 7.2 shows the performance improvement after incorporating the modifications in *Cost-Greedy* algorithm. For a query location, if we consider all the points in its first quadrant then, we need to examine a lot of plans to check whether their costs are within the permissible threshold. But, if we only look at immediate neighbour points to a query locations, the number of plans reduces substantially.

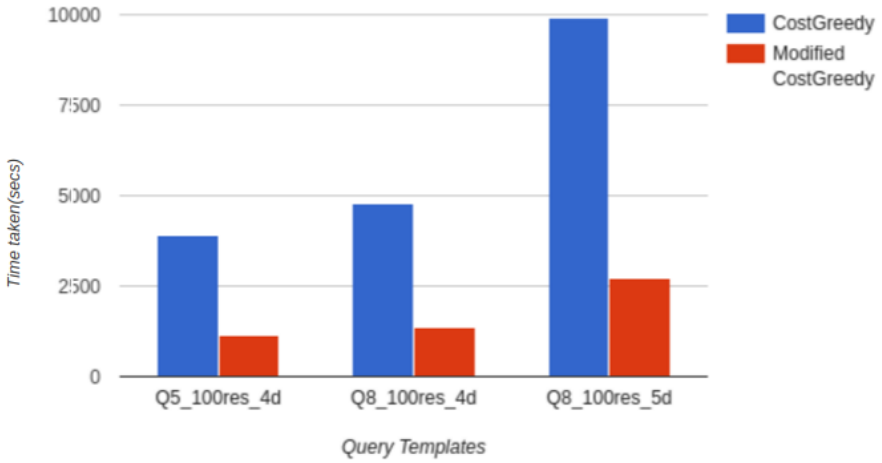


Figure 7.2: CostGreedy performance

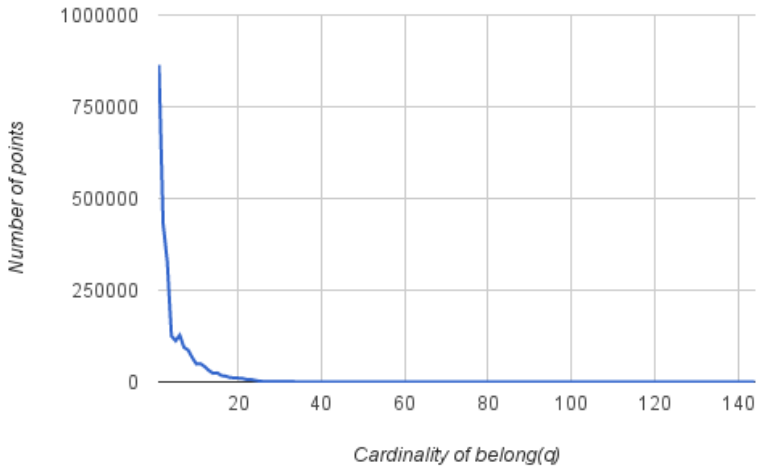


Figure 7.3: Number of points with different cardinality of $belong(q)$

For example, the cardinality of POSP of TPC-H query 8 with 4 dimensions and 40 resolution is 145. But, figure 7.3 depicts that for majority of the points, cardinality of $belong(q)$ is very small compared to the total number of plans. Thus, we can achieve such computational efficiency.

7.4 Plan Reduction Quality

We have produced the POSP set for different TPC-H query templates by varying each dimension uniformly. Table 7.1 shows the cardinality of POSP sets for different query templates. For example, TPC-H query 8 with 2D ESS and 100 resolution has 25 plans in its POSP set.

Query Template	#plans
Q8_100res_2d	25
Q10_100res_2d	18
Q7_100res_3d	45
Q8_100res_3d	71
Q8_100res_4d	215
Q8_100res_5d	573

Table 7.1: POSP cardinality for different query templates

These results show that the cardinality of the optimal plan set can reach high values for some queries. Also, as we increase the dimensions, we notice a substantial increase in the number plans even with fixed resolution.

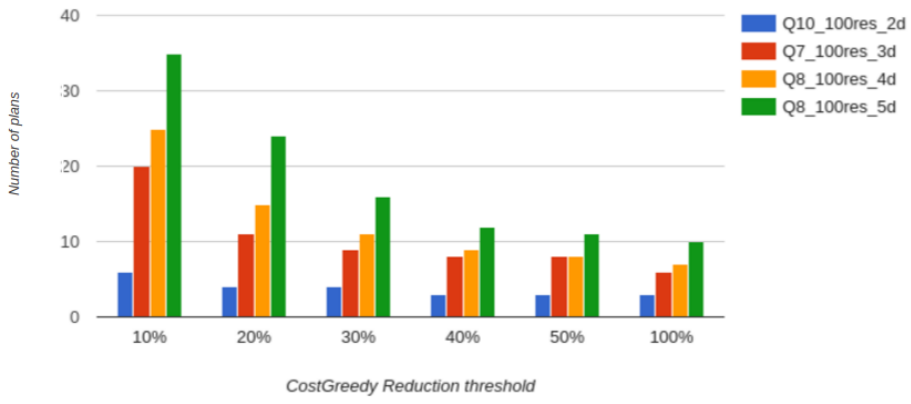


Figure 7.4: Plan reduction quality by varying dimensions

Now turning our attention to reduction quality, figure 7.4 shows that *CostGreedy* significantly reduces plans in POSP set. We have also found that 80% of the plans in the reduced

plan diagrams could not be swallowed in the reduction process. Another observation is that the initial steep exponential decrease in the number of plans with increasing threshold. We have found this to be true as stated in [3] even with high resolution. But, with increase in dimensions, the number of reduced plans goes beyond "*anorexic*"level (small absolute number of plans). The reason for this is that *CostGreedy* follows a conservative cost-bounding approach to estimate the costs of plans outside their endo-optimal regions. Plan reduction quality can be improved by using $CostGreedfy - FPC$ algorithm [4]. The *foreign-plan-costing (FPC)* feature is used to evaluate plans outside of their endo-optimal regions.

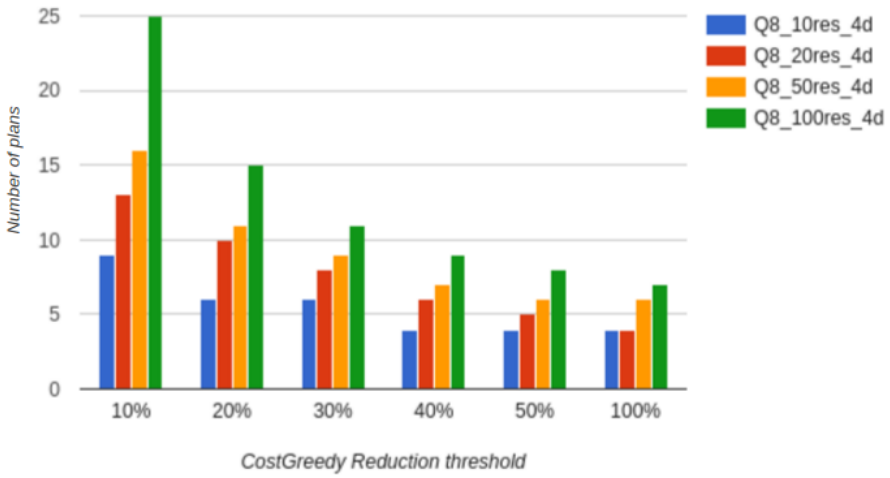


Figure 7.5: Plan reduction quality by varying resolution

Figure 7.5 shows that the final plan cardinality increases with the resolution. As we increase the resolution, new plans are found near the axis which can not be swallowed by other plans. This is the reason that the final plan cardinality increases. We can use *foreign-plan-costing (FPC)* feature to check whether the final plan cardinality can be reduced further.

Chapter 8

Conclusions and Future Work

In the initial work, we have generalized the implementation of Quest so that any query with any number of error-prone selectivity predicates can be evaluated. As part of Quest modifications, we have implemented *join selectivity injection* feature in PostgreSQL 9.4.
In our second contribution, we have proposed a parallel algorithm for identifying POSP set for a query template. We have shown that our algorithm improves the identification time from *years* to *hours*. We have also improved the running time of *CostGreedy* reduction algorithm. Finally, we investigated the possibilities of reducing a dense plan diagram produced by PostgreSQL, without adversely affecting the query processing quality. Our analysis has shown that plan reduction can be carried out efficiently and can bring down the plan cardinality to a manageable number of plans while maintaining acceptable query processing quality.
To further improve the plan reduction quality, we can use *CostGreedy-FPC* algorithm. But, as stated in [4], the algorithm performs very poorly in terms of running time. It takes mn FPC calls to the optimizer to reduce a ESS with m points and n plans. It will be an interesting future work to reduce the running time of *CostGreedy-FPC* by exploiting parallelism using multi-core architectures. It also opens up the possibility to analyze plan bouquet technique and its empirical bounds for high-dimensional queries.

Bibliography

[1] A. Dutt and J. Haritsa. *Plan Bouquets: Query Processing without Selectivity Estimation.* In SIGMOD, 2014. ii, 1, 2, 3, 4

[2] A. Dutt, S. Neelam, and J. Haritsa. *QUEST: An Exploratory Approach to Robust Query Processing.* In PVLDB, 2014. ii, 2, 7

[3] Harish D., P. Darera, and J. Haritsa. *On the Production of Anorexic Plan Diagrams.* In VLDB, 2007. ii, 2, 8, 17, 24

[4] Harish D. Pooja N. Darera Jayant R. Haritsa. *Identifying Robust Plans through Plan Diagram Reduction.* In VLDB, 2008. 24, 25

[5] Jayant R. Haritsa. *The Picasso Database Query Optimizer Visualizer.* In VLDB, 2010. 3, 17

[6] Mohammad Aslam. *Picasso: Design and implementation of a Query Optimizer Analyzer.* Master's Thesis, Dept. of Computer Sci. and Automation, IISc, (2006) . 12

[7] C Rajmohan. *Turbo-charging Plan Bouquet Identification.* Masterś Thesis, Dept. of Computer Sci. and Automation, IISc, (2014) .

[8] http://www.jfree.org/jfreechart. 8

[9] http://www.jgraph.com. 8

[10] http://www.tpc.org/tpch. 20

[11] http://www.tpc.org/tpcds.

[12] http://www.serc.iisc.in/facilities/cray-xc40-named-as-sahasrat/. 20