SpillBound Implementation in QUEST

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

Computer Science and Engineering

BY

Davinder Singh

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

June, 2017

Declaration of Originality

I, **Davinder Singh**, with SR No. **04-04-00-10-41-15-1-12055** hereby declare that the material presented in the thesis titled

SpillBound Implementation in QUEST

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2015-2017**.
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 discussions.
- 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

Acknowledgements

I would like to express my sincere gratitude to my project advisor, Prof. Jayant R. Haritsa for giving me an opportunity to work on this project. I am thankful to him for his valuable guidance and moral support. I feel lucky to be able to work under his supervision.

I am thankful to Srinivas Karthik for his assistance and guidance. His suggestions helped me a lot. This project would not have been possible without his constant support and motivation. I also sincerely thank my lab mates for constant motivation and support to put all the ideas into reality. I am overwhelmed to acknowledge their humbleness. Also I thank my CSA friends who made my stay at IISc pleasant, and for all the fun we had together.

I would also like to thank the Department of Computer Science and Automation for providing excellent study environment. The learning experience has been really wonderful here. Finally, I am indebted with gratitude to my parents for their love and inspiration that no amount of thanks can suffice.

Abstract

For a given SQL query, modern database system recommends an *optimal* plan for its execution based on selectivity estimates. These estimates are often in error due to inaccurate statistics and invalid assumptions made by database systems, thus leading to highly inflated query response times. The *sub-optimality* incurred for these executions, due to erroneous estimates, even reach the millions! How good is the plan depends on the fact how accurately database system can estimate selectivities. Recently, PlanBouquet [1] and *SpillBound* [2], tries to addresses this problem by jettisoning the selectivity estimations, instead, discovering the actual selectivities at run time incrementally through a sequence of cost-limited executions of carefully chosen plans.

The goal of this work is to implement SpillBound algorithm over PostgreSQL which is an open-source relational database system. The implementation requires code modification in PostgreSQL, to support the *sub-plan executions* and *selectivity monitoring* features. Furthermore, we have integrated the above mentioned implementation into QUEST tool [3], which is an prototype system of PlanBouquet approach [1]. QUEST provides interactive interface and user control while query execution for SpillBound technique. Additionally we have built a framework to measure the sub-optimality of a query for native database query Optimizer and integrate it into the QUEST system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this Chapter, we present necessary details about query processing in relational database systems and the selectivity estimation problem in these systems.

1.1 Query Processing in Relational Database Systems

Data processing in relational database systems is achieved using SQL queries. In relational database system for each SQL query, query Optimizer module considers different ways or *plans* to execute the query and chooses estimated least expensive plan in terms of resource consumption. Query Optimizer module is the brain of database system because this is where the whole *planning* about query execution is done. Let us now consider an example benchmark query, which can be seen below.

```
SELECT * FROM lineitem, orders, part
WHERE p_partkey = l_partkey and
l_orderkey = o_orderkey and
p_retailprice < 1000 and l_extendedprice < 2000;
```

The above example query consist of two filter predicates which are ($p\_retailprice < 1000$) and ($l\_extendedprice < 2000$) and two join predicates which are ($p\_partkey = l\_partkey$) and ($l\_orderkey = o\_orderkey$).

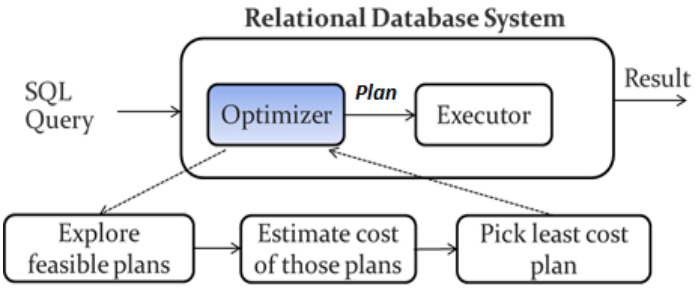


Figure 1.1: Query Processing

Figure 1.1 represents the main internal modules of database system that are responsible for query optimization. We can see in *Figure 1.1* that, database system will pass given SQL query to Optimizer module for constructing a *plan*, where *plan* is an ordered set of steps to access and process data.

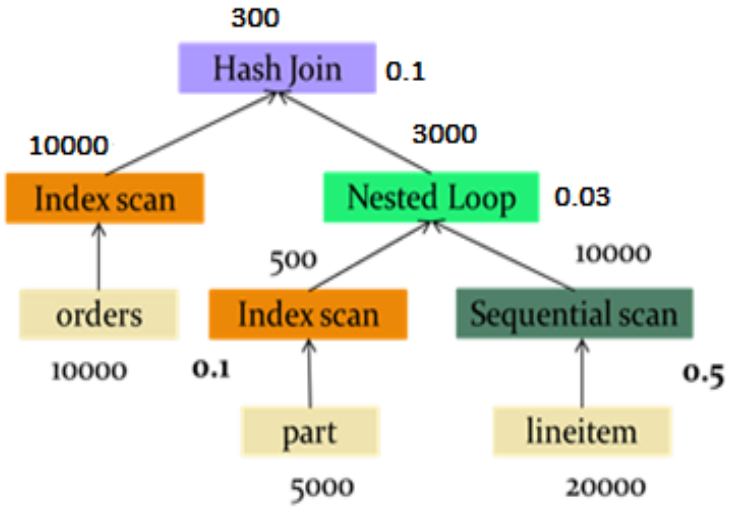


Figure 1.2: Query Plan Tree

Figure 1.2 shows an example plan which can used for executing the above query. Constructing an optimal plan for a query is non-trivial because for constructing an optimal plan Optimizer has to explore all feasible set of plans then estimate the cost of those plans and pick the least *cost* plan, where *cost* is a measure of query response time. The problem is that the Optimizer cannot know the exact cost of a plan without executing it, so it estimates the cost of each plan. The estimated cost is primarily a function of *predicate selectivities*, where predicate selectivities is a fraction of output no. of tuples of the predicate to the worst-case output no. of tuples. The selectivity is measured at each step (or node) of the plan. The Opti-

mizer has to estimate the selectivity of each node to identify the ideal plan for query execution. These estimates are based on various factors like summary statistics and assumptions such as attribute-value independence.

1.2 Selectivity Estimation Problem

Due to inaccurate statistics, or coarse summaries or complex user predicates, these selectivity estimates are often erroneous leading to highly inflated query response times. The *sub-optimalities* incurred for these executions, due to erroneous estimates, even reached millions!

To handle this issue recently *PlanBouquet*[1] and *SpillBound*[2] have been proposed which works in a different way than the above mentioned approach i.e., instead of estimating these selectivities before execution of a plan, it incrementally discover the actual selectivities through closely monitoring partial executions of carefully chosen set of plans. For realization of this approach we need inbuilt selectivity monitoring and sub-plan techniques in underlying database system.

The goal of this work is to design and implement all techniques that are required for realization of SpillBound. Further we have designed a graphical user interface tool to visualize its performance as compared to native database systems and with other related techniques such as [1].

1.3 Organization

In rest of this document, Chapter 2 provides the preliminaries and overview of the SpillBound algorithm. Further, in Chapters 3,4,5 and 6 we have discussed our contribution in details like modification of source code of open source database system. Thereafter, we have discussed about the QUEST tools and its features in Chapter 7. In the last we have concluded our contribution.

Chapter 2

Overview of SpillBound

In this Chapter, we have explained the new robust query processing technique SpillBound[2].

2.1 Preliminaries

As we saw, database Optimizer estimates selectivites of number of nodes to get the ideal execution plan for the query execution. In practice because of number of other factors, these estimates differ from actual selectivities that we eventually get after query execution. Such erroneous selectivities are called Error-Prone Selectivities and the query predicates that are responsible for these error-prone selectivities are called Error-Prone Predicates(EPP). It is safer to assume all predicates of the query are prone to errors but for making it easy to understand in following SQL query we will assume only two join predicates are error-prone.

```
SELECT * FROM lineitem, orders, part
WHERE p_partkey = l_partkey and
l_orderkey = o_orderkey and
p_retailprice < 1000 and l_extendedprice < 2000;
```

These error-prone predicates are used to create Error-prone Selectivity Space(ESS). This space is approximately discretized fine grained grid which represents all possible combinations of selectivities of error-prone predicates. *Figure 2.1* represents 2-dimensional ESS whose each dimension represents one error prone predicate of example query. Each dimension ranges from [0,1] representing selectivity of the respective predicate.

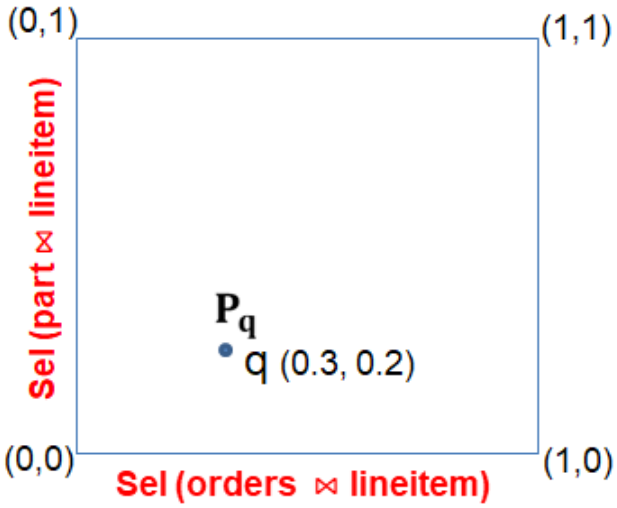


Figure 2.1: Error-prone Selectivity Space

Each location $q \in [0,1]^D$ in ESS represents a query with corresponding selectivity values. Here D represent the number of dimensions in the ESS. For example point *q(0.3, 0.2)* represent query with selectivity of predicate ***l_orderkey*** = ***o_orderkey*** as ***0.3*** and selectivity of predicate ***p_partkey*** = ***l_partkey*** as ***0.2*** in 2-dimensional space. Each point in ESS stores plan P_q and information about the plan like cost of the plan for the given selectivities of EPPs.

2.2 Overview of SpillBound[2] Algorithm

SpillBound is a new query processing algorithm which completely avoids selectivity estimation process and discovers the actual selectivity explicitly at run time through a sequence of partial executions of carefully chosen plans. In this whole discussion SpillBound assumes *Plan Cost Monotonicity(PCM)* property to hold true. PCM states that cost of plans increases with increase in selectivities of error-prone predicates.

For each point in ESS, it is also assumed to get the optimal plan (from the Optimizer) corresponding to the selectivity values at that point. Once we have this information for every point in ESS, then we identify *Isocost Contours* on the ESS generated. Isocost contour is a notion which represent a set of locations in ESS whose optimal plan's cost is the same. Lets assume C_{min} is the minimum cost and C_{max} is the maximum cost in ESS. These isocost contours follows *contour-doubling regime* i.e., cost of any contour is $2^k * C_{min}$, where k = 1,2, ... $\left\lceil \log_2 \left(\frac{C_{max}}{C_{min}} \right) \right\rceil$ represents contour number. *Figure 2.2* shows hyperbolic isocost contours that results from a 2D ESS.

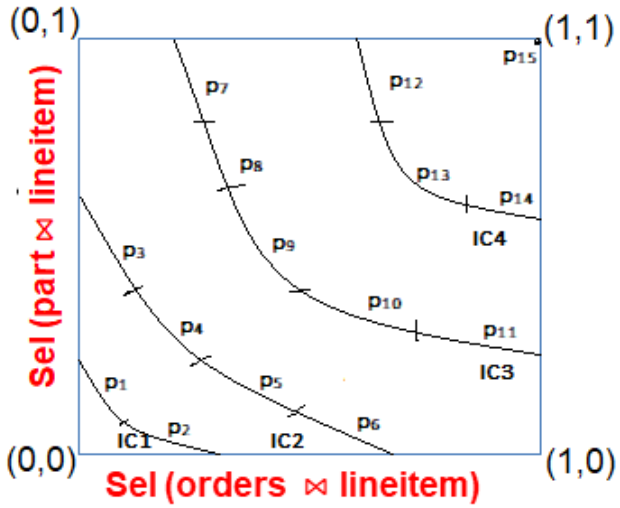


Figure 2.2: ESS with isocost contours

Till this point, the whole process of generating contours in ESS is called compile-time query processing in *SpillBound*. Before proceeding to run time execution of query we need to understand how the existing database system executes a plan in query execution phase. The Executor module of existing database system executes a plan generated by Optimizer module for input query in a *pipelined* manner. In a query plan, the concurrent execution of a contiguous sequence of plan nodes is called a *pipelined* manner execution and these concurrently executing set of nodes constitute a *pipeline*. *Figure 2.3* shows how each node in a pipeline passes qualified tuples to its parent node concurrently and parent node does the same till the end of the pipeline.

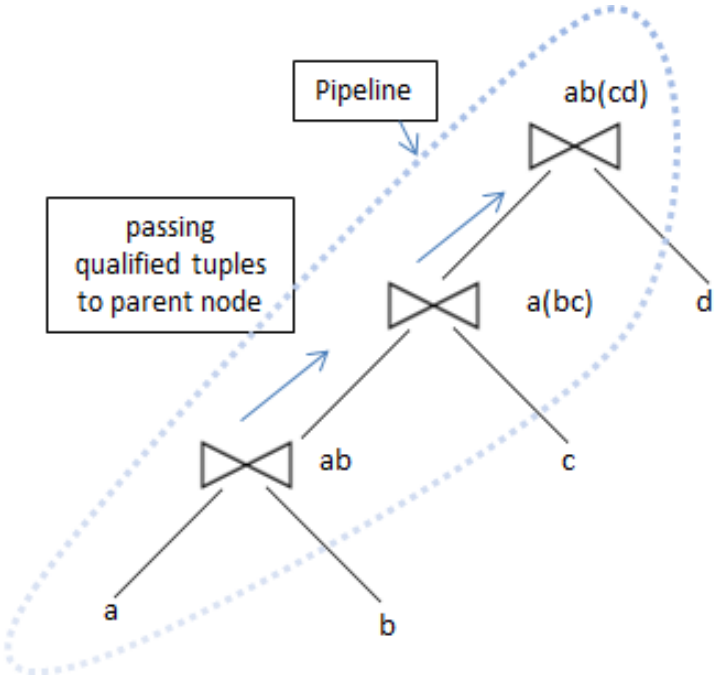


Figure 2.3: Execution of Pipeline in a Plan

In pipeline execution we assume that only one pipeline is under execution at any time i.e., a single thread is executing for a plan, which seems to be true even in current database systems.

At run time *SpillBound* explicitly discover the selectivities for error-prone predicates by leveraging the notion of *spilling*. *Spilling* involves modifying the way Executor executes a plan to extract the increased learning about the selectivities within the *assigned execution budget*. Here executing a query within the assigned execution budget means we will be setting time or cost for the execution of query before it start execution and after completely consuming this budget Executor will stop the execution of query irrespective of whether Executor finishes the query execution or not.

In order to better explain this we are using the same following example from SpillBound[2]. For expository convenience, in a plan tree for an internal node, the nodes that are in the sub-tree rooted at that node are called its upstream nodes, and the nodes that are on its path to the root are called its downstream nodes. Suppose we are interested in the selectivity of an EPP e_j. Let the internal node corresponding to e_j in plan P be N_j. The key observation here is that the execution cost of N_j's downstream nodes in P is not useful for learning about the selectivity of N_j. So, discarding the output of N_j without forwarding to its downstream nodes, and focusing the entire budget to the sub-tree rooted at N_j, helps in learning more selectivity of e_j. So after *spilling* (dropping without forwarding) the output of a node (EPP) increases the selectivity learning at that node (EPP) in the plan tree. Following diagram shows the modified

execution of a plan in *SpillBound*.

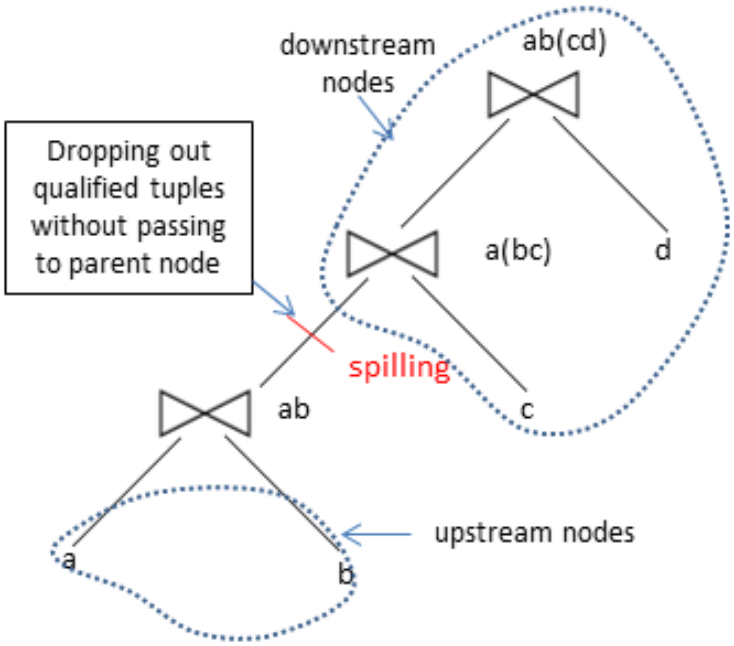


Figure 2.4: *Spilling* the output of a node in a Plan

Chapter 3

Our Contribution

We now tern our attention to contribution of this work which describes required mechanisms to materialize the SpillBound and integrate this implementation with QUEST[3], by providing the graphical user interface to execute the query. Here, we have provided the brief introduction about implementation of these mechanisms.

This implementation is completed in following two phases.

- Implementation of SpillBound Algorithm as a driver program.
- Customizing open source PostgreSQL9.4.1[4] database engine.

3.1 Implementation of SpillBound Algorithm as a driver program

In this phase of implementation, we have implemented SpillBound algorithm in java which includes identification of all the *pipelines* in the plan and then identify spilling node in plan. More about this section we have explained in Chapter 5.

3.2 Customizing open source PostgreSQL9.4.1 database engine

For supporting the SpillBound algorithm we need some extra features in the underlying database system. For this purpose we have modified the source code of open source PostgreSQL database system. Here, we have provided brief introduction about these features.

Spilling-Mode of Execution: This feature enables database system to execute a plan in spill mode i.e., a given plan will be executed till the specified node in the plan after which it will stop the execution of the plan. More information about this feature is in Chapter 4.

Selectivity Monitoring while plan execution: This feature enables us to monitor the selectivity of a predicate in the given plan while its execution. This feature is explained in more details in Chapter 6.

One of the already existing feature that we used is **Cost bounded plan execution:** user can set limit on time or cost of any query before its execution, such that after the limit database engine will stop execution and return the output produced till that point of time.

Above mentioned phases are completed in following manner while focusing on the main features of the algorithm. Details about these features are given in following chapters.

- What is *Spilling* and *Spilling-Mode of execution*?
- Given a plan and set of *error-prone predicates(EPPs)*, which EPP should be the spilling predicate?
- How to monitor the selectivity of the spilling predicate?

Chapter 4

Spilling-Mode of execution

In Chapter 2 we saw the benefit of spilling the output of EPP node in a plan while execution. Generally Optimizer passes a plan to executor for execution and executor will execute that plan till its root node. For *spilling* on certain node(EPP) of plan this time we have to pass a plan and a node(EPP) whose output we want to spill. Now we have to stop execution at the spilling node so that it does not pass the result to its parent node. For this purpose we will pass only spilling node(EPP) rooted sub-tree to the executor as a plan. To implement this feature we modified the executor module of open-source database system PostgreSQL[4].

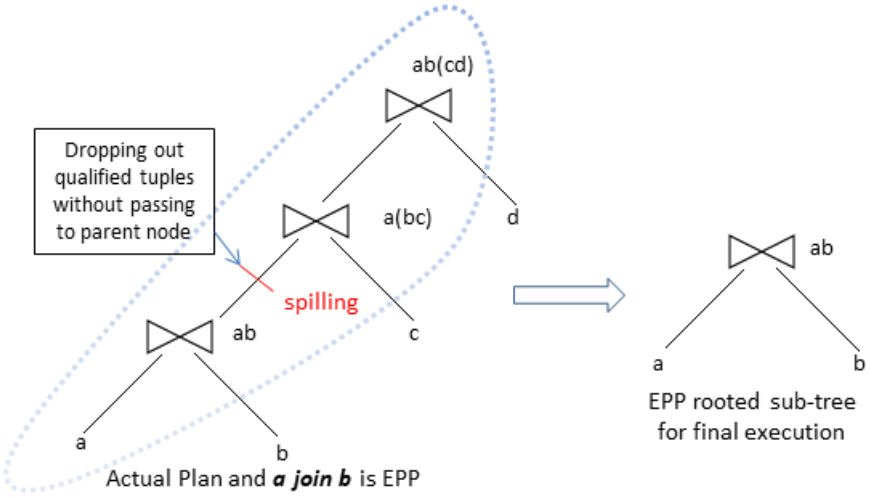


Figure 4.1: Proposed implementation of *spilling*

Spilling-Mode of execution of a plan is shown in above diagram.

The SpillBound driver program give an input plan and a node (node representing error prone predicate in the plan as spilling node) as input to the underling modified database system. In

return it will get the output of the spilling node rooted sub-plan. Actually, we modified the database system in such manner that it stop the execution of the plan at the spilling node.

Spilling-Mode execution feature implementation design is shown in following diagram.

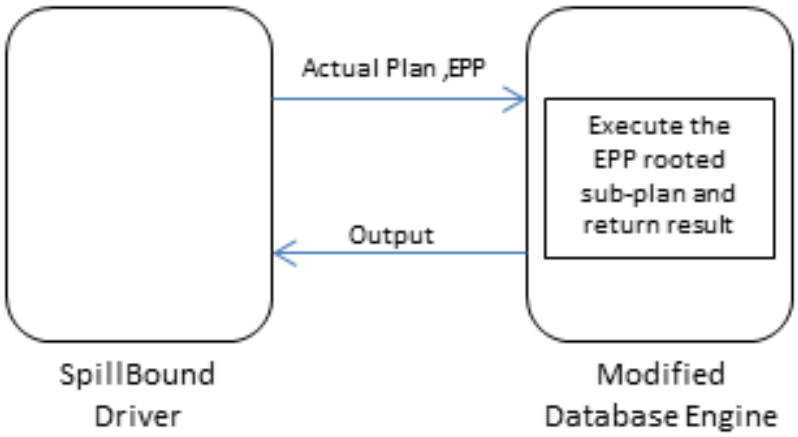


Figure 4.2: Implementation of *spilling* in Database System

Spill-mode execution of a plan could also be achieved by giving spilling node rooted sub-plan as the only input to the database system, but in this case we need to modify the database system in such manner that it allows sub-plan of given query to execute as final plan for the query. Both of the above cases seems to be same but here, difference is in first case, we are tricking the database system by stopping the execution at certain node which results as executing a sub-plan whereas in second case we are forcing database system to execute a sub-plan i.e., execute a plan for the query which is not a plan recommended by the Optimizer for the input query. So to implement the spill-mode execution by second method requires huge modification in source code of database system as compare to the implementation of first case because in this case we need to disable all the checks that are done by the optimizer to ensure that the recommended plan is valid plan for the input query.

Still the ideal way to implement spill-mode execution is, generate query from the sub-plan that we want to execute in such a manner that, when next time we give that query as input, database system chooses the same plan for execution from which we have generated that query. In this way we don't have to modify the database system. But this is not trivial because this process is reverse of the non-trivial task of generating a plan for a query in modern database systems.

Chapter 5

Spill Node Identification

As we have discussed in Chapter 2 that in conventional database query processing, the execution of a query plan can be partitioned into a sequence of pipelines as in [5]. Intuitively, a pipeline can be defined as the maximal concurrently executing sub-tree of the execution plan. The entire execution plan can therefore be viewed as an ordering on its constituent pipelines. So now for given a plan and a set of error-prone predicates, which predicate should be chosen for spilling so that we get more information about the selectivity. We have discussed about this in following section.

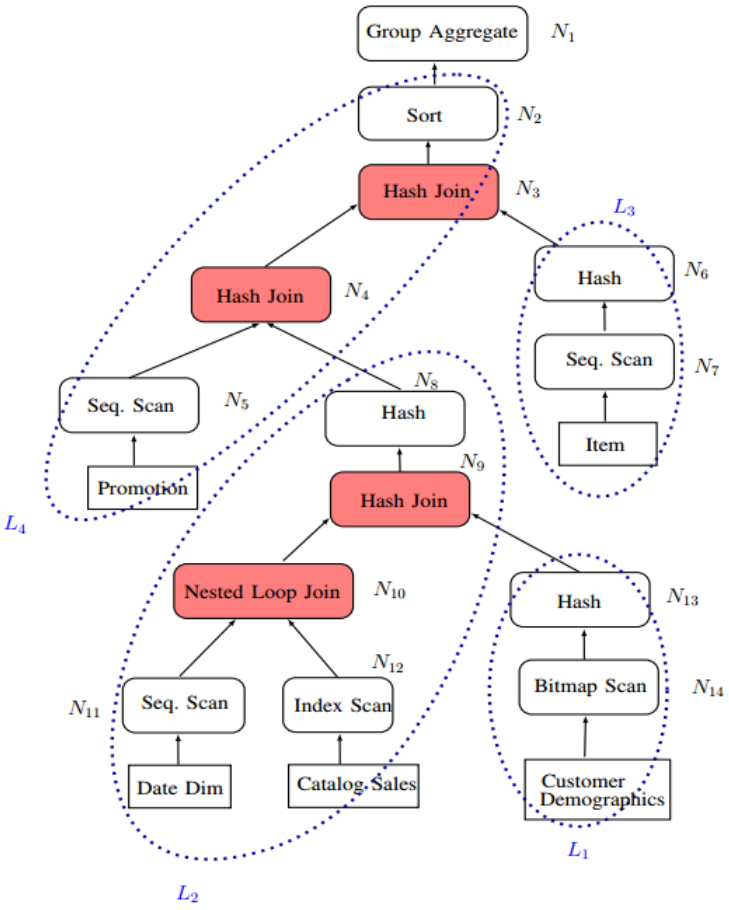
5.1 Which *EPP* should be the *spilling predicate*?

The reason for asking this question is the case when we have more than one error-prone predicate in the plan. So now we have given a plan and a set of error-prone predicates, which predicate should be chosen for spilling so that we get more information about the selectivity. This is done based on following two rules: **Inter-Pipeline Ordering:** Order the EPPs as per the execution order of their respective pipelines.
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.

Following is the example of identification of spilling EPP in a plan given in *figure 5.1*.

- Identify the pipelines in plan as in [5].
  $\{L_4,\ L_3,\ L_2,\ L_1\}$
- Identify pipeline order based on engines execution order. In *Figure 8* since L_4 is ordered after L_2, the EPP nodes N_3 and N_4 are ordered after N_9 and N_{10}.
  $\{L_1,\ L_2,\ L_3,\ L_4\}$

- Find total order of EPP based on intra-pipeline ordering and inter-pipeline ordering.
  $\{[N_{10},\ N_9],\ [N_4,\ N_3]\}$
- Choose first EPP from above order.
  $\{N_{10}\}$



Execution Plan Tree of TPC-DS Query 26

Figure 5.1: Pipeline Identification in a Plan

We have implemented this algorithm as part of SpillBound driver in java.

Chapter 6

Selectivity Monitoring

Since we have a mechanism to find the order in which EPPs should be arranged for spilling in a plan and execute that plan in spilling-mode, the next challenge is to measure the selectivity that we learned during this execution which is explained in following section.

6.1 Monitoring the selectivity of the spilling predicate

For measuring the selectivity of EPP, we categorize the predicates as follow.

Filter Predicate : This type of predicates involves only one table in predicate expression as shown in *figure 6.1*.

For example $p\_retailprice < 1000$ is one filter predicate. Selectivity for these type of predicates can be calculated as follow.

$$selectivity = \frac{output\ cardinality}{input\ cardinality}$$

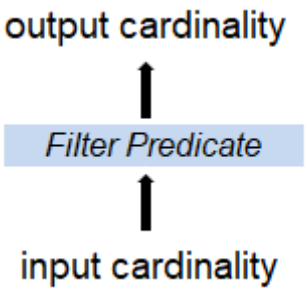


Figure 6.1: Filter Predicate example

Join Predicate : Predicate expressions involving two tables are called join predicates. For example predicate $p\_partkey = l\_partkey$ involves *part* and *lineitem* relation of TPC-H[6] standard benchmark schema. Following diagram represents a join predicate.

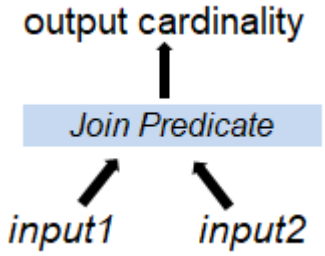


Figure 6.2: Join Predicate example

Join Predicates are further categorized to following two types.

PK-FK join predicate: The predicate expressions in which two tables are joined over a common column or an attribute which is the primary key of one of the table and the other column involved in the join is the foreign key in the other table and that foreign key is referencing the primary key of first table.

$$selectivity = \frac{output\ cardinality}{foreign\ key\ cardinality}$$

Non PK-FK join predicates: In this case two tables are joined over non key column or attribute.

$$selectivity = \frac{output\ cardinality}{input1\ *\ input2}$$

We use above formulas to monitor the selectivity of the error-prone predicate. As we can see for computing the selectivity we need some input values to these methods. In order to get those input values and compute the selectivity of the EPP we have modified the Executor module of the open-source Postgres[4] database system. *Figure 6.3* shows complete process of executing one plan in spilling mode and monitoring the selectivity of EPP. Important point to keep in mind is, while computing selectivity for any predicate, we should have complete information about denominator. i.e., node which is giving cardinality for the denominator should be completely executed otherwise we may get inaccurate selectivity.

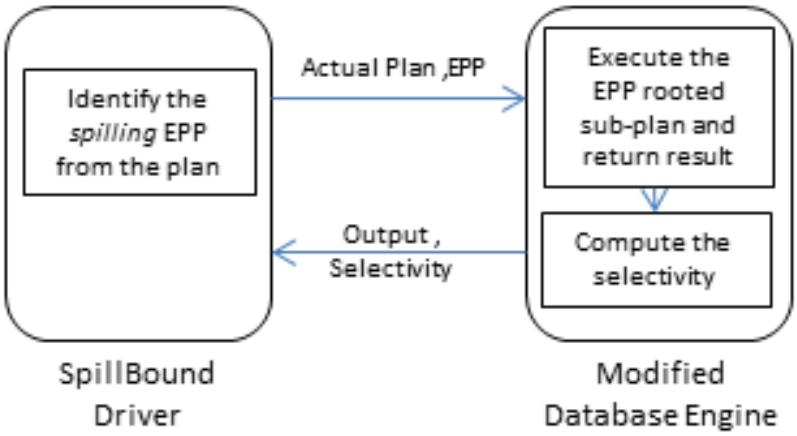


Figure 6.3: SpillBound Algorithm for executing one plan

All database systems push filter predicate to the leaf nodes of the plan tree i.e., filter predicates are computed before the join predicates. Since each plan execution will start by executing filter predicates so for filter predicate we don't have to worry about incomplete denominator information because its input is complete table and we can get the information about the size of table from statistic about data stored by database system. But for the join predicates we ensure this by choosing first EPP from the total order of EPPs in a plan while identifying spilling EPP in a plan in Chapter 5. In this way we will always choose the lower node of plan as spilling node. Still there are cases when filter predicate is computed before join predicate(EPP) and it takes output of filter predicate as input. Because of pipelined execution while computing selectivity there is incomplete denominator information problem. We can avoid this problem by waiting till the execution is finished and to keep track of selectivity learned in between use estimated output of filter predicate as input to the join predicate.

To ensure that we are using correct methods to calculate the selectivity as the PostgreSQL[4], we did following sanity check. We executed the standard benchmark TPC-H[6] query on PostgreSQL[4] and get the *annotated plan*. Annotated plan represents a plan which contains information that we get after query execution means we will also have the information about the number of rows we got after the complete execution of every node. With the information from annotated plan we computed the selectivity for each node and then we used these selectivity values as input to the database system PostgreSQL[4] with the same query for second time execution. Here giving selectivity values as input means instead of estimating the selectivity of predicates Optimizer module of PostgreSQL[4] database system uses input selectivity as the actual selectivity. In second execution we are taking only the optimal plan recommended by the Optimizer and this plan may differ from the previous plan because this time plan is generated

using the input selectivity values instead of using estimated selectivity values. Our goal is to verify that our selectivity computation methods are same as PostgreSQL database system. So again if we get the same selectivity from the estimated plan implies that we are using the same method as of the Postgres[4] to compute the selectivity.

Special case for selectivity monitoring: When spilling predicate is filter over Index Nested Loop join operator. Generally, Postgres database system evaluate all filter predicates before any join predicate of the query. But in case of Index Nested Loop(INL) join it evaluates join before filter predicate in following manner.

```
-for each tuple(row) in outer relation(table)
    -find all matching tuples using index
     in inner relation (join)
    -check the filter predicate condition
     on matched tuples (filter)
    -pass result to parent node
```

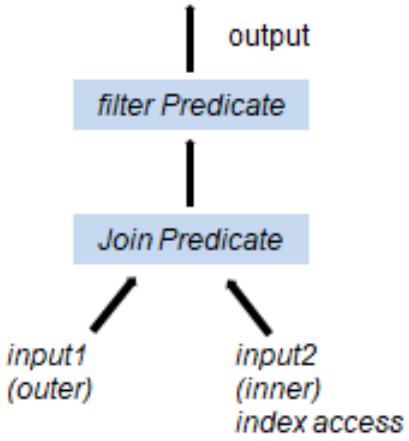


Figure 6.4: Filter Over INL join

Figure 6.4 shows the conceptual idea about the filter over the INL join. While computing the selectivity for filter predicate we encountered following issues:

- Complete information about intermediate cardinality will be known only after complete execution.
- Intermediate cardinality results information is not accessible to user.

Only solution for the first problem is we should not calculate the selectivity till filter predicate is completely evaluated because only then we will have complete information about the denominator.

Second problem is resolved by adding a counter variable after the first step (join) to see how many rows are being filtered out in the first step of the Index Nested Loop. Now we have total rows after complete Index Nested Loop and rows after the first step so we can get the selectivity of the filter predicate over Index Nested Loop. All of the above process is done by modifying the open-source Postgres[4] database system.

Chapter 7

QUEST

QUEST (QUery Execution without eSTimation) is a prototype implementation of *PlanBouquet*[1]. It provides a graphical user interface with implemented *PlanBouquet* algorithm. It visually shows query execution using PlanBouquet technique and provides interactive interface during execution of a query. It also provides performance visualization of native Optimizer and PlanBouquet. QUEST interface is implemented using java swing. Quest feature that we have implemented is explained in more detail as follows.

7.1 Computing Query sub-optimality

In QUEST[3] this feature was present but it was only able to get the sub-optimality of query whose error-prone predicates are only filter predicates. Here query sub-optimality is the ratio of the costs of plan chosen by a database system using the estimated selectivity values to plan generated using the actual selectivity values. We have extended this feature to the join predicates as well i.e., now we can get the sub-optimality of query in which join predicates are also error prone. Getting the actual selectivity values of predicate in the query is not enough because sometimes Optimizer can infer predicates from the existing predicates in the query. To understand this we need to understand how Optimizer choose a plan for a Query. We are going to use following hypothetical example SQL query to understand how optimizer generate plan for any query.

```
Select ...
From R1, R2, R3, R4
Where R1.x = R2.x and R2.x = R3.x and R3.x =
R4.x;
```

Above query consist of four tables (R1, R2, R3, R4) and three join predicates (R1.x = R2.x

and R2.x = R3.x and R3.x = R4.x) where **x** represent any attribute of the joining tables. We are using a chain query just to get more inferred predicates. We know that Optimizer pushes the filter predicates to the leaf level so only thing that is important to generate an optimal plan is optimal join order.

Relational Database System Optimizer uses the Selinger's Algorithm[7] to choose optimal join order. The core idea behind this algorithm is the following.

```
Start from n = 1 to total number of relations in query.
a.  Find best join order of n relations among all possible join orders of
    n relations.
b.  To find best join order for n+1 relations, join previously calculated
    join order in step (a) with each additional relation separately.
c. goto step (a).
```

Above algorithm is explained using *figure 7.1* taken from Optimality slides of Shivnath Babu.

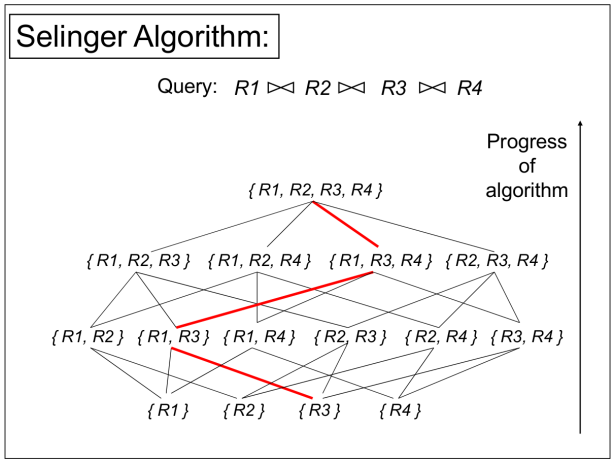


Figure 7.1: System R Algorithm

As we can see in above diagram, Optimizer starts search for join order from bottom level, and going till the level equal to the number of tables in query. At each level it consider all the possible combinations of join orders and chose the best and then proceed to the higher level with the best join order chosen at current level. For identifying the best join order optimizer needs to know the actual cost of each join order of that level which is not an easy task. Finally at the top level Optimizer will find the best join order, which will be used for generating the final plan of the query.

Query Number	PostgreSQL optimized Plan	Optimal Plan	Sub-optimality
Q1b(19)	776 ms	1.6 ms	485
Q1d(19)	756 ms	3.0 ms	240
Q13d(139)	84 sec	64 sec	1.3

Table 7.1: Query Sub-optimality

Now we know how optimizer generate an estimated optimal plan for a query. Similarly to get an optimal plan we need the actual selectivity values of all the nodes in above diagram, this can be achieved only by executing all of them, which is very expensive.

Even making estimation of selectivity values of all nodes is not easy, so to make the life of Optimizer easier most of the database system vendors uses following assumption.

> Selectivity of all join predicates is independent of each other.
> or
> Selectivity of a join predicate will remains the same irrespective of the level in plan at which it is getting computed.

With this assumption, database optimizer's work reduces significantly because now it only has to estimate the selectivity of nodes at the first and second level of *figure 7.1* and for the rest of the levels it can use the same selectivity.

If we use the same assumption to find the actual selectivity values of the predicates our work will reduce significantly. Now we have to compute the selectivity of nodes which are at the first and second level in *figure 7.1*. This is because each node in this diagram represents a join predicate and due to the independent behavior of each of the predicate, the selectivity remains same. We have implemented this feature by modifying both Optimizer and Executor modules of open-source PostgreSQL database system.

Following table shows the sub-optimality that we got for the queries taken from Join Order Benchmark (JOB)[8] on underlying Database System PostgreSQL 9.4.1[4].

7.2 QUEST Architecture

Existing Quest architecture is shown in *figure 7.2* which consists of various modules. Introduction about each module is as follow.

User Interface, It provides graphical interface to user for giving a query as input and display a pop-up window showing all predicates and we can select error prone predicates from that set of predicates. **Parser** takes input SQL query from user interface and parse it. Parsing

given query includes validating the query and identifying the predicates that input query contains. **PlanBouquet**[1] is a robust query processing technique that was already implemented in QUEST. **Performance Visualization** shows the performance comparison of database system PostgreSQL, PlanBouquet algorithm and SpillBound algorithm with respect to the ideal database system which already knows about the selectivity values of all predicates. Next module is Modified open source PostgreSQL 9.4 database system. It incorporate all the feature that are required to make recently proposed techniques work.

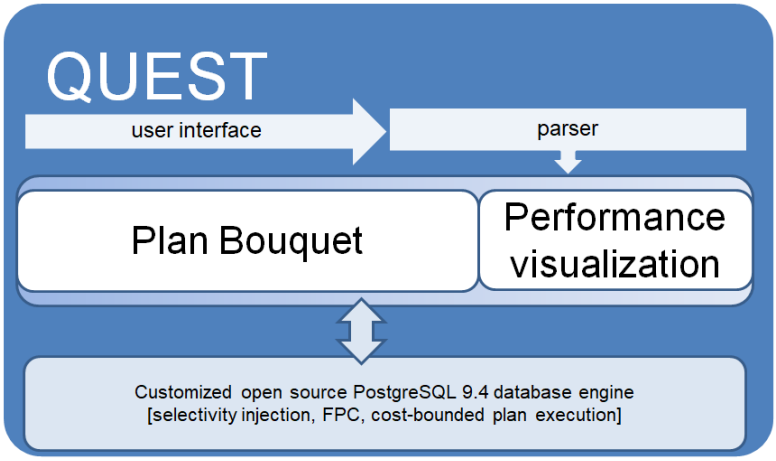


Figure 7.2: Existing QUEST Architecture

Figure 7.3 represents the modified QUEST architecture. we have integrated the SpillBound algorithm as we can see in the following diagram.

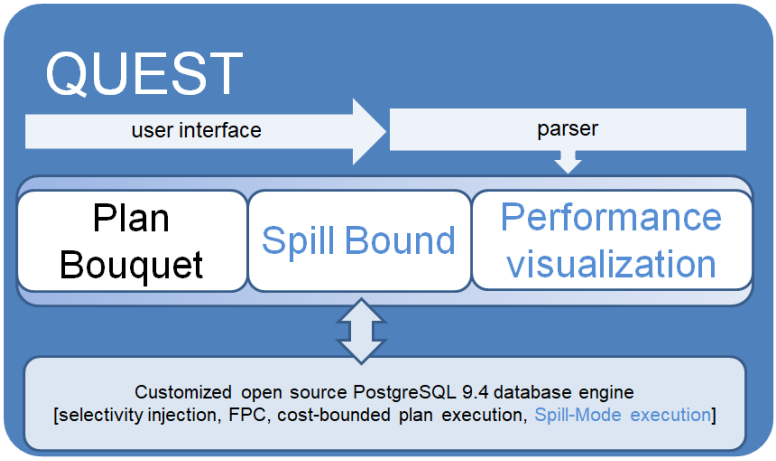
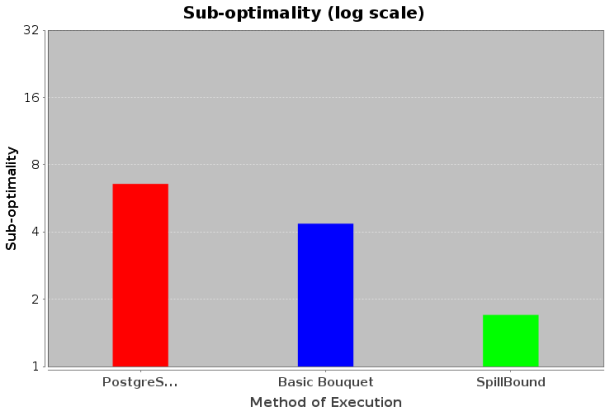
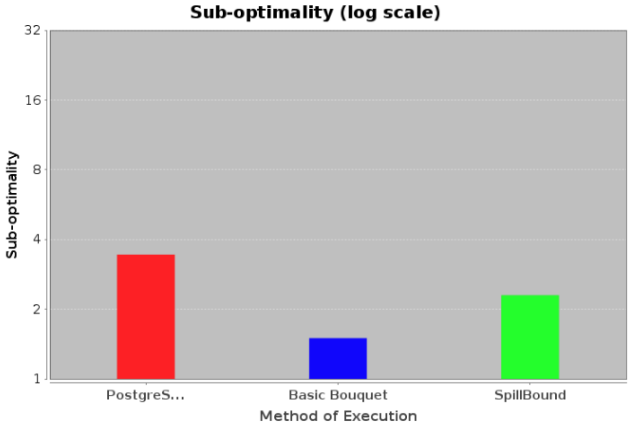


Figure 7.3: Upgraded QUEST Architecture

We already discussed about *Query sub-optimality* and this work is used for enhancing the



(a) TPC-H Q19



(b) TPC-DS Q91

Figure 7.4: Performance Visualization

performance visualization feature. Figures in 7.4 are produced using performance visualization feature of QUEST on underling database system PostgreSQL9.4.1. Fig. 7.4a is showing the performance of query Q19 from TPC-H[6] schema on 1GB database on PostgreSQL database system, plan bouquet and SpillBound. Similarly fig. 7.4b is generated for query Q91 from TPC-DS[9] schema on 100GB database. For both the queries we assume only two predicates are error-prone predicates.

Now we move to our main feature in this implementation i.e., SpillBound execution. It illustrate the discovery of selectivity values of error prone predicates during the partial execution of carefully chosen set of plans from cheapest isocost contour to higher isocost contours. *Figure 7.5* shows the SpillBound Execution interface for QUEST.

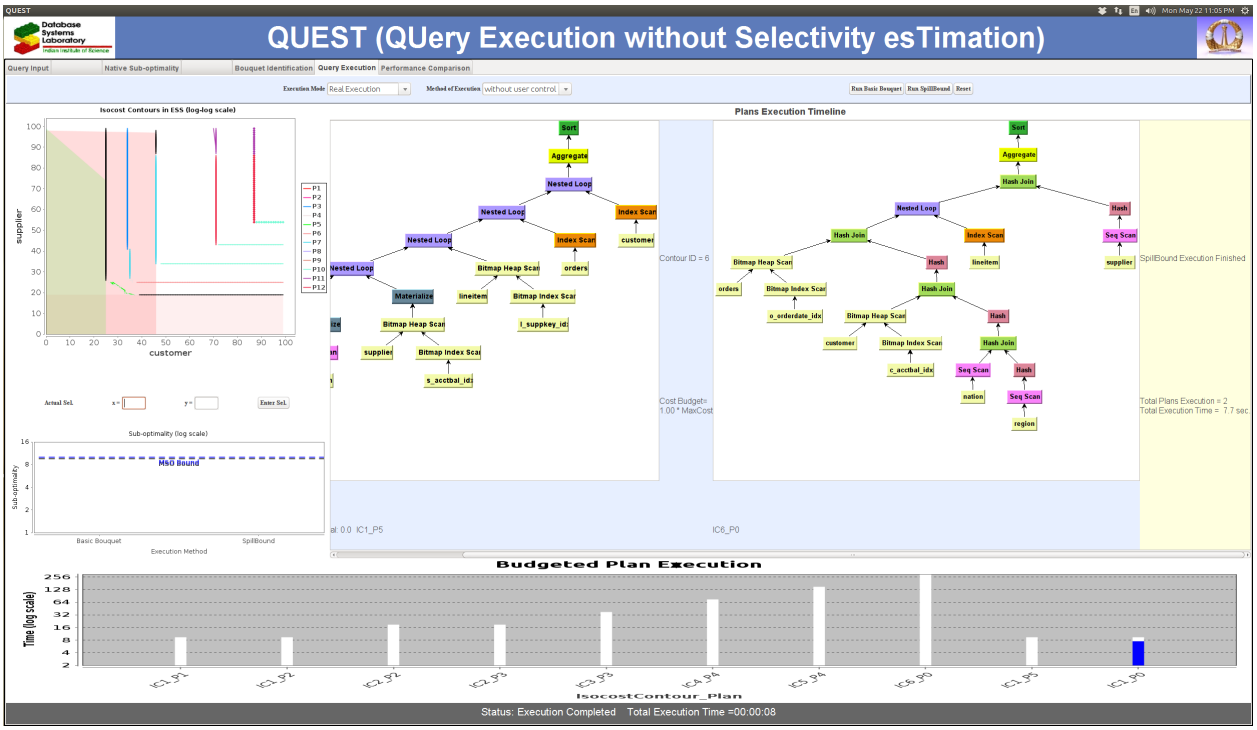


Figure 7.5: SpillBound Execution interface for QUEST

Chapter 8

Conclusions and Future Work

In this document we have discussed implementation details about various modules of SpillBound algorithm as driver program and the features that are required in underling database system to materialize it. This driver program also support running query with extra control like stopping execution after each plan execution. With this feature after one plan execution, query execution can be stopped until the next execution instruction received. Further we have integrated SpillBound algorithm to QUEST prototype system and shows the performance of the SpillBound compare to native database system and PlanBouquet[1]. Important future work is as follows.

- Implementation of further enhanced version of SpillBound algorithm called AlignedBound[2] algorithm and integrate it to the QUEST.
- Testing the performance of both the algorithms using Optimizer Torture Test(OTT) proposed in [10].

Bibliography

[1] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: Query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1039–1050, New York, NY, USA, 2014. ACM. ii, 3, 19, 22, 25

[2] S. K. Venkatesh, J. R. Haritsa, S. Kenkre, V. Pandit, and L. Krishnan. Platform-independent robust query processsing. *IEEE Transactions on Knowledge and Data Engineering*, PP(99):1–1, 2017. ii, iii, 3, 4, 5, 7, 25

[3] Anshuman Dutt, Sumit Neelam, and J. R. Haritsa Haritsa. Quest: An exploratory approach to robust query processing. *Proc. VLDB Endow.*, 7(13):1585–1588, August 2014. ii, 9, 19

[4] The postgresql global development group. `https://www.postgresql.org/ftp/source/v9.4.1/`. 9, 11, 16, 17, 18, 21

[5] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM. 13

[6] Tpc benchmark h standard specification revision 2.17.1. `http://www.tpc.org/tpch`. 15, 17, 23

[7] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM. 20

[8] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015. 21

[9] Tpc benchmark ds standard specification revision 2.5.0. http://www.tpc.org/tpcds. 23

[10] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1721–1736, New York, NY, USA, 2016. ACM. 25