Sub-query Based Approach for Robust Query Processing

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Technology

IN

Computer Science and Engineering

BY

Gourav Kumar

Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

June, 2018

Declaration of Originality

I, **Gourav Kumar**, with SR No. **04-04-00-10-42-16-1-13302** hereby declare that the material presented in the thesis titled

Sub-query Based Approach for Robust Query Processing

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2016-2018**.
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

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 also sincerely thank Dr Anshuman Dutt without his collaboration and constant motivation this work would not have been possible.

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 would like to thank all IISc staff, my lab mates, my family and friends for helping me at critical junctures and making this project possible.

Abstract

Selectivity estimates play a critical role in SQL query optimization. They often (mis)lead the query optimizer to produce highly sub-optimal execution plans. Previously, “Plan Bouquet” [2] approach demonstrated that it is possible to guarantee upper bounds on execution sub-optimality – surprisingly, by choosing to discard compile-time estimates and discovering the selectivity values at run-time. Existing techniques based on this approach rely on an optimization-intensive preprocessing phase to deliver the promise of bounded sub-optimality, making them unsuitable for ad hoc queries. In addition, these techniques make selectivity independence assumption. Selectivity independence assumption says that the joint selectivity of a conjunction of query predicates is given by the product of their individual selectivities. All relational database engines make this assumption in their query optimizer and, is also one of the reasons for their sub-optimality.

In this work, first, we argue that the preprocessing phase is not an integral part of plan bouquet approach and is actually wasteful in ad hoc query scenarios. We analyze and implement a revamped plan bouquet algorithm that retains the ability to provide sub-optimality guarantees with low optimization overhead. This is done by executing sub-queries with only one unknown join predicate in a cost budgeted manner. Second, we also exploit the topological structure of the input query graph to derive significantly improved sub-optimality bounds. Experiments with TPC-DS benchmark queries show that, compared to previous techniques, optimization overheads are enormously reduced and there is virtually no loss in the empirical sub-optimality performance. Finally, we extend this approach to work without selectivity independence assumption and also, give query graph dependent worst case sub-optimality bounds for it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Query optimizer is a critical component in relational database systems. Its job is to convert a declarative SQL query into a procedural execution plan that takes minimum time to obtain the query results. To produce the optimal execution plan query optimizer needs accurate estimates of the selectivities for query predicates. As these selectivity estimates are mostly error-prone, query optimizer often produces suboptimal execution plans. Figure 1.1 shows an example SQL query. It has two base predicates $p\_retailprice < 1000$ and $s\_acctbal < 95$, and three join predicates ($p\_partkey = l\_partkey$ (PL), $s\_suppkey = l\_suppkey$ (SL) and $o\_orderkey = l\_orderkey$ (OL)). To get the optimal plan for this query, optimizer needs accurate estimates of all its predicates. Since base predicate selectivities can be estimated quite accurately using metadata statistics, in this work, don't consider them error-prone and we focus only on the join predicates. Also, a join predicate is error-prone if its selectivity cannot be accurately estimated using information already stored within a database, e.g. $o\_orderkey = l\_orderkey$ in the given query is not error-prone.[1]

```
Select * From part P, lineitem L, supplier S, orders O
Where P.p_partkey = L.l_partkey and S.s_suppkey = L.l_suppkey and O.o_orderkey =
L.l_orderkey and P.p_retailprice < 1000 and S.s_acctbal < 95
```

Figure 1.1: Query Q1

Earlier there were no bounds on sub-optimality of plans produced due to an error in selectivity estimates. To deal with the sub-optimality due to inaccurate selectivity estimates, a

[1]The selectivity of this predicate can be accurately estimated as the inverse of tuple count of *orders* relation since there is no base-relation predicate on *orders* and it joins key attribute of *orders* with foreign-key attribute of *lineitem*.

novel query processing approach called "Plan Bouquet" [2] was proposed. Plan bouquet approach skips selectivity estimation and rather discovers the actual selectivities. In fact, it could also provide upper bound on maximum sub-optimality (MSO) that can occur due to additional effort spent in selectivity discovery. Later [3] proposed a modification to plan bouquet based approach resulting in improved sub-optimality bounds of the form $D^2 + 3D$, where D is the number of error-prone selectivities in the query. This was achieved by doing a cost budgeted execution of sub-queries with only one unknown predicate. SpillBound [3] requires a compile-time processing of each selectivity location in the error-selectivity space (ESS), to identify the plan to be executed for each subquery. ESS consists of an axis for each error-prone predicate, denoting its selectivity which varies from 0-1. The actual selectivity of the query could lie anywhere in the ESS. Optimizing each point in the ESS requires many optimization calls at compile-time. The number of optimization calls is equal to resolution of the ESS raised to power number of error-prone predicates, this could easily go up to millions. Hence, these techniques could not be used in ad hoc query scenarios. Also, in the proposed form, both techniques continued to depend on independence assumption to determine selectivities for predicate combinations and hence the resulting MSO bounds are not valid when the assumption does not hold true.

The selectivity independence assumption says that the joint selectivity for a combination of query predicates is given by the product of individual predicate selectivities. But since the assumption does not hold in practice, it leads to erroneous selectivity estimates and hence suboptimal execution plan. For the above query, the estimated selectivities of all predicate combinations can be computed by estimating the selectivities of PL and SL followed by application of the selectivity independence assumption. For example, the joint selectivity of PL and SL, which is required to compute the size of an intermediate relation where only P, L, and S are joined, can be computed by using selectivities of PL and SL by using the selectivity independence assumption.

But this estimate may be wrong if selectivity independence does not hold. In short, the joint selectivity of (PL and SL) together is an additional unknown to be estimated if selectivity independence assumption is not made. Further, in case OL is also an error-prone join selectivity, the number of unknowns with independence assumption increases from 2 to 3, but it increases sharply from 3 to 7, when independence assumption is not utilized.

Our Contributions In this work, first, we discuss an algorithm for executing queries without any compile-time overhead proposed in [1]. This algorithm works in a way similar to SpillBound, by doing cost budgeted execution of sub-queries with one unknown error-prone join predicate. But, unlike SpillBound we don't require any compile-time preprocessing to identify a plan for it. Instead, we identify the plan for any such sub-query at run-time using two modules cost budgeted planning (CBP) and selectivity learning potential (SLP). At a high level, CBP module takes a query and cost budget, and returns the optimal plan for that query, having cost equal to the required cost budget. SLP module takes a plan with one unknown predicate and a cost budget and returns the selectivity that can be learnt for that predicate using the given plan for that cost budget.

Then, we analyze the worst case sub-optimality guarantees (MSO) for this algorithm. In fact, we show that for chain query graph (query graph has a node for each relation and an edge for each join predicate, it is discussed in Section 2) MSO is linear in D, that is, $8D - 6$, matching the lower bound $O(D)$ shown in [3]. This is achieved by exploiting the topological structure of the given query graph. More generally, we derive a generic expression for MSO as a function of D and $\mathcal{K}$, where $\mathcal{K}$ represents the extent of query graph complexity, i.e., its closeness to star-graph. We also implement the approach and compare its empirical performance with SpillBound. The empirical results show the MSO remains almost comparable, with a significant decrease in optimization overheads. In fact, the optimization overheads are in a few hundred compared to millions in the previous approaches.

Finally, we extend this approach to work without selectivity independence assumption and provide a worst-case sub-optimality analysis for it. The basic idea is to utilize the very fact that there is some interaction between the actual selectivities of the individual join predicates and their joint selectivities. For instance, the actual selectivity of predicate PL would not give any information regarding actual selectivity of predicate SL, but any of these values would give an upper bound on the joint selectivity of PL and SL together. Also, we exploit the observation that all combinations of query predicates may not even be possible. For example, for the query shown in Figure 2, we do not need the joint selectivity of PL and OC since no intermediate query result directly applies these two predicates together.

Select * **From** part P, lineitem L, orders O, customer C
Where P.p_partkey = L.l_partkey and O.o_orderkey = L.l_orderkey and O.o_custkey = C.c_custkey and P.p_retailprice < 1000 and C.c_acctbal < 105

Figure 1.2: Query Q2

Q1 is an instance of star-shaped join graph queries and Q2 is an example of chain shaped join graph queries. We prove that for any query D join predicates connected in star-shaped join graph, MSO bound is given by $5 \times 2^{D-1}$ while if they are connected in chain shaped join graph, the MSO bound is much lesser at $D^2 + 3D$. Surprisingly, it turns out that the MSO bound for chain shaped query graphs remains exactly same as the MSO bounds with independence assumption (as reported in [1]).

Organization We start by giving problem framework in Section 2. We discuss an algorithm for handling ad hoc queries with independence assumption in Section 3. Then in Section 4, we prove its MSO guarantees and analyze the MSO equation. Algorithm implementation and experimental analysis are done in Sections 5 and 6 respectively. In Section 7, we relax selectivity independence assumption and give an algorithm for executing queries. This is followed by its worst-case analysis for different shapes of query graphs in Section 8. Finally, we conclude in Section 9 along with directions for future work.

Chapter 2

Problem Framework

Any Select-From-Where SQL query can be represented as a Join Graph. **Join Graph** consists of a node for each relation and there is an edge between two nodes if the corresponding relations have a join predicate between them. If the join between two relations is a Key-Foreign Key join, then the edges will be directed edges pointing towards the key side. The join graph for query Q1 (in Fig. 1.1) is shown in Fig. 2.1.

A join graph is called star-shaped join graph (star graph for short) when there is a central relation and all other relations are connected to the same central relation with a join-predicate, e.g., join graph for Q1. On the other hand, a chain graph is one where any relation is connected to at most two relations, one on each side, and thus resembles the shape of a chain, e.g. join graph for query Q2 (Fig. 1.2) would look like $P - L - O - C$.

Each connected component of the join graph is referred to as an **intermediate relation** and represents the possible intermediate results for different execution plans for the query. Fig. 2.2 shows the base relations, intermediate relations and the output relation for the join graph in Fig. 2.1.

For each of the intermediate relations, we can construct an **Intermediate Query** by selecting the required relations and predicates from the original query text. Fig. 2.3 shows intermediate query for intermediate relation PSL.

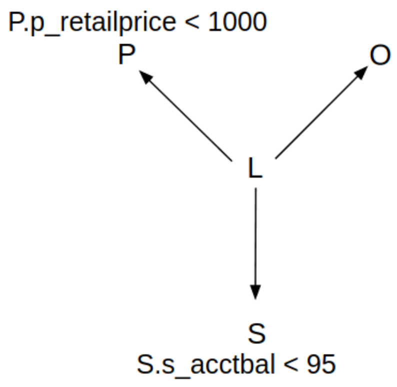


Figure 2.1: Join Graph for Query Q1

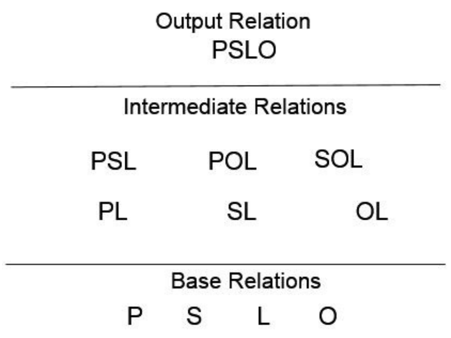


Figure 2.2: Intermediate relations for Query Q1

```
Select * From  part P, lineitem L, supplier S
Where P.p_partkey = L.l_partkey and S.s_suppkey = L.l_suppkey and
P.p_retailprice < 1000 and S.s_acctbal < 95
```

Figure 2.3: Intermediate query for PSL

For the original problem of computing the query result for output relation, each intermediate relation can be termed as a subproblem. For each such subproblem, the output cardinality can be represented by a function of the cardinality of participating base relations and selectivities of the corresponding predicates. This function is called as **Cardinality expression**. The exact function depends on whether the join is key-foreign key join, generic inner join or outer join.

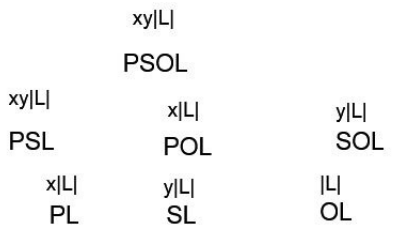


Figure 2.4: Cardinality expression for subproblems in Q1

For query Q1, the cardinality expression for all its subproblems is shown in Fig. 2.4. x and y denote the unknown selectivities of join predicates $P \bowtie L$ and $S \bowtie L$ respectively. Notice, that join predicate for $O \bowtie L$ is not error-prone as the join is Pk-Fk join and there is no base predicate on key side O. The output cardinality of $O \bowtie L$ is equal to $|L|$ (where $|L|$ denotes cardinality of relation L).

Dimension of a subproblem is the number of unknowns in its cardinality expression. Subproblems PL and SL are in 1-D (dimension), whereas PSL and PSLO are in 2-D. If we construct the ESS for PSL it will have two dimensions, x and y. We use D to denote the number of dimensions for the query.

Chapter 3

Algorithm for Ad Hoc Queries

Subproblems with same unknowns p in cardinality expression, are clubbed into a set Q^p. Q^ϕ is the set of all subproblems with known cardinality. For query in Fig. 1.1, the set of subproblems will be $Q^\phi = \{OL\}$, $Q^x = \{PL, POL\}$, $Q^y = \{SL, SOL\}$, $Q^{xy} = \{PSL, PSOL\}$. For each 1-D set we use cost budgeted planning (CBP) and selectivity learning potential (SLP) module to identify a plan to be executed.

Cost-Budgeted Planning The module takes the query Q and a cost value C_{pick} as input, along with selectivity bounds q_{lb} and q_{ub} (By default q_{lb} is 0 and q_{ub} is 1) such that $C_{opt}(q_{lb}) < C_{pick} \leq C_{opt}(q_{ub})$. The aim is to identify an intermediate selectivity location, denoted with q_{pick}, such that $q_{lb} < q_{pick} < q_{ub}$ and $C_{opt}(q_{pick}) = C_{pick}$, and then returns the plan $P_{opt}(q_{pick})$. A 1-D subproblem will have only one optimal plan for a cost budget C_{pick}.

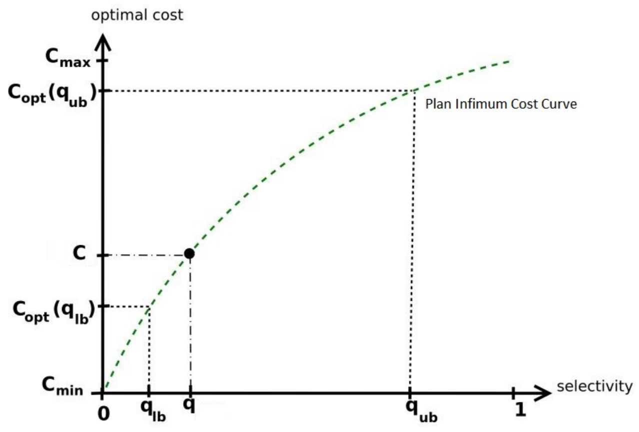


Figure 3.1: Cost-budgeted planning

We use one more module called Selectivity Learning Potential (SLP), in the algorithm.

Given a 1-D query, the SLP module takes a plan P and cost budget C, and returns the selectivity that can be learnt by executing plan P with cost budget C. Given, at least two plans and a cost budget for a dimension, max SLP refers to the plan which can learn maximum selectivity for that dimension in the given cost budget. e.g., in Fig. 3.2, $P1$ and $P2$ are max SLP plans for cost budget C_1 and C_2.

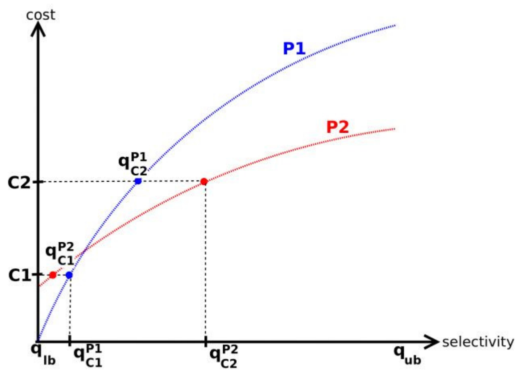


Figure 3.2: Shows max SLP plan for C_1 and C_2

Finding max SLP plan using Cost-Budgeted Planning The algorithm works by doing only one execution per 1-D subproblem set. To identify the plan for a 1-D subproblem set, we use CBP and SLP modules. CBP is invoked on each subproblem in the 1-D set, which returns one plan for each subproblem. These subproblems are given to SLP module along with a cost budget, which returns the max SLP plan among them. Max SLP plan for dimension x, is identified by invoking CBP on subproblem PL and PLO, and then using max SLP on the plans returned (shown in Fig 3.3). The max SLP plan corresponds to PLO.

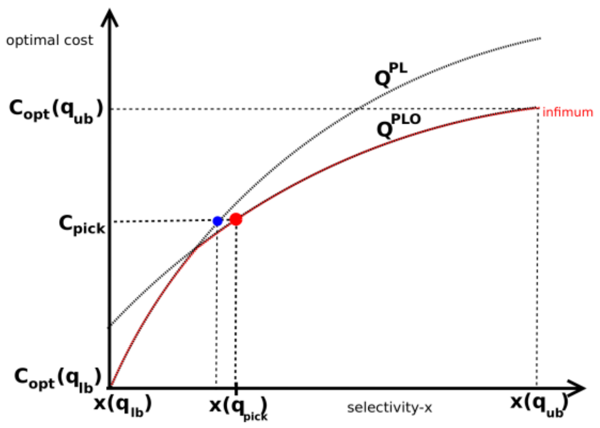


Figure 3.3: Identification of max SLP plan using CBP and SLP

Algorithm 1 AQUA (Ad hoc Query Algorithm) [1]

```
1: $q_{lb} = (0, 0, ..., 0)$;
2: $q_{ub} = (1, 1, ..., 1)$;
3: $D_e = x_1, x_2, ..., x_D$
4: $k = 1$;
5: complete = false;
6: while complete == false do
7:     Restart:
8:     for $x_i$ in $D_e$ do
9:         $Q^{x_i}$ = findIntermediateQueries($x_i$);
10:        $P^k_{x_i} = CBP(Q^{x_i},\ x_i(q_{lb}),\ x_i(q_{ub}),\ cost(IC_k))$;
11:        execInfo = $CBE_{WB}(P^k_{x_i}, cost(IC_k))$;
12:        status = execInfo.status;
13:        $selec_{x_i}$ = execInfo.learnedSelec;
14:        if status == terminated then
15:            $q_{lb} = update(q_{lb}, selec_{x_i})$;
16:        else
17:            $q_{lb} = update(q_{lb}, selec_{x_i})$;
18:            $q_{ub} = update(q_{ub}, selec_{x_i})$;
19:            $D_e = D_e - x_i$;
20:            reorganize intermediate query partitions and find revived dimensions;
21:            goto Restart;
22:        end if
23:    end for
24:    if $D_e$ is non_empty then
25:        $k++$;
26:    else
27:        complete = true;
28:    end if
29: end while
30: execute $P_{opt}(q_{ub})$;
```

The algorithm starts by finding C_{min} and C_{max} values for the query. Where C_{max} is the cost of executing the query when all predicates have max selectivity. Similarly, C_{min} corresponds to cost of the query, when all predicates are at min selectivity. This cost range is divided into $m(= log_2(C_{max}/C_{min}))$ budgets, where cost of k^{th} budget IC_K is given by $C_{max}/2^{m-k}$. The problem is executed with cost budget IC_1, if it doesn't terminate, some minimum cardinality is learnt. Then we move to the next cost budget. The algorithm has following parts

Choosing next 1d subproblem In this algorithm, we do not worry about the order of execution of 1-D subproblems. In experiments, we discuss an approach using which we decide the order.

Processing a 1d subproblem For each 1-D subproblem set, we invoke CBP and SLP and get max SLP plan. This plan is executed with the given cost budget, and the selectivity learnt is updated. By using the max SLP plan we ensure that we learn the maximum selectivity for that dimension, within the given cost budget.

Feedback from 1d subproblem After cost budgeted execution, if the subproblem terminates successfully, then we learn the exact selectivity for that dimension and its q_{ub} is updated. Otherwise, some lower bound on selectivity is learned, which is passed on as feedback by the executor. Thus, q_{lb} is updated.

Reorganize problem structure When a subproblem completes execution, it may reduce subproblems at other levels to 1-D, thus some new subproblems are added to 1-D sets. If one such 1-D set has already terminated execution then it needs to be executed again for the same cost budget, this is called as **Revival**.

Consider query in Fig. 1.1, the 1-D subproblem sets for it are $Q^x = \{PL, POL\}$ and $Q^y = \{SL, SOL\}$. Initially, $q_{lb} = (0, 0)$ and $q_{ub} = (1, 1)$. Lets say we execute a plan for Q^x identified using CBP and SLP, this plan terminates within the cost budget. Hence, we don't learn x completely, but learn some lower bound for it. Similarly, for y also we learn some lower bound. Now, as all dimensions have terminated we move to next cost budget, and q_{lb} is updated to say $(0.2, 0.3)$.

Let's consider another scenario, say after termination of the plan for dimension x, we execute the plan for dimension y. Plan for dimension y completes, and thus we know the selectivity value for y. Now, due to completion of y dimension, PSL and $PSOL$ will become 1-D and will be added to the Q^x subproblem set. As new subproblems have been added to set Q^x, thus even if it failed for the same cost budget earlier, we will have to execute the plan for it again. Which can also be said as subproblem set Q^x was revived due to completion of dimension y. Revivals are bad as we have to execute the same subproblem set again for the same cost budget, which could have been avoided if set Q^y executed first.

3.1 Maximum Number of Executions With No Revivals

We point out two crucial facts regarding our decomposition approach i.e.,

1. Any 1-D execution requires only one execution per cost-budget.

2. The number of 1-D subproblems is exactly D.

Now, the importance of completed 1-D executions is already evident from the fact that each such execution has the ability to reduce the dimensionality of the original problem one by one, finally reaching a 1-D problem that requires only 1 execution per cost budget. Next, we present the generic result that establishes the ability of terminated executions to make progress in solving a D-dimensional problem.

Result 1 For a given query Q with multiple error-prone selectivities, if the execution created using a 1-D subproblem Q^x for a budget cost(IC_k) is terminated, then any plan for Q that uses an intermediate relation from the set Q^x cannot complete execution of Q within the budget cost(IC_k).

Proof: Since the execution with maximum learning potential for Q^x and budget cost(IC_k) is terminated, we know a lower bound on the actual selectivity of x. The lower bound on x is such that all the intermediate relations in Q_x have cost more than cost(IC_k). As a result, any plan for Q that uses one or more intermediate relations from Q^x costs more than cost(IC_k) and hence cannot complete within budget cost(IC_k).

Unlike completed execution, a terminated execution does not have the permanent benefit of reducing the dimensionality and hence leads to wasted overheads in the execution sequence. Any execution plan for Q must use at least one intermediate relation from the 1-D subproblems of Q, as we are considering binary joins only. Hence, the following result holds true:

Lemma 1 For a query Q and budget cost(IC_k), if the executions are terminated for all the 1-D subproblems, then the optimal cost for Q is certainly more than cost(IC_k).

Maximum Number of 1-D executions to cross a cost budget Here, we prove results regarding the number of executions for a given cost-budget, using the above lemma:

Result 2 For a query with D error-prone dimensions, the maximum number of terminated executions with a fixed cost-budget are D.

Proof: Since the initial number of 1-D subproblems is D and if all executions are terminated then none of the subproblems is revived in the process.

While the sequence of only terminated executions is bounded by D, the total number of 1-D executions for any cost budget can still be more than D if revivals are allowed. This is because a completed execution may lead to revival of the subproblems for which a terminated execution has already been performed. The maximum number of executions due to revival depend on the query graph structure, this is analyzed in the next section.

3.2 Comparison With SpillBound

SpillBound and Aqua both work on similar lines. For each dimension, they pick a plan to be executed. The difference lies in the way a plan is chosen for execution. While SpillBound chooses to spill on the plans present on the contour, Aqua optimizes the intermediate query for each dimension using CBP and SLP modules. Now the way Aqua picks a plan it will not miss a plan considered by SpillBound for any dimension. SpillBound has information based on the whole query optimization from which it will execute the subplan, whereas Aqua will take intermediate query for that dimension and optimize it. Becuase of optimizing the intermediate query we may get a better plan for that dimension for that cost budget. Which means that for the same cost budget we could learn more selectivity than SpillBound but not less than it. Hence, MSO of Aqua will be at least as good as that of SpillBound.

Chapter 4

Join Graph Specific Analysis

Let's consider join graph given in Fig. 4.1 (Fig. 4.2 shows subproblems for it). Assume that all its dimensions are error-prone. Let's say after some cost budgeted executions QR dimension is known. The set of subproblems with unknown cardinality reduce to as shown in Fig. 4.3, here the subproblems in curly braces have the same dimension which is due to the completion of dimension QR. This is similar to saying that essentially that the edge QR has shrunk in join graph(i.e. the two nodes are merged, shown in Fig. 4.4). The number of revivals due to completion of QR is equal to the number of neighbouring edges it has (Here it will be 4).

Let x denotes selectivity of join predicated represented by edge QR in join graph. From the above example, we can observe two points:

1. The maximum number of revivals due to completion of dimension x is equal to the number of neighbouring edges QR has

2. The effect of completion of a dimension x on the set of subproblems is same as if the nodes of that edge QR have merged

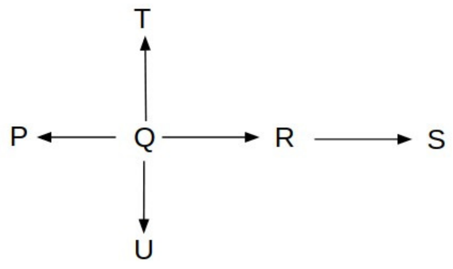


Figure 4.1: Join Graph before merging Q and R

PQRSTU
PQRST PQRSU PQRTU QRSTU
PQRS PQRT PQRU PQTU QRST QRSU QRTU
PQR PQT PQU QRS QRT QRU QTU
PQ QR QT QU RS

Figure 4.2: Subproblems before QR is known

PQRSTU
{PQTU, PQRTU} PQRST PQRSU QRSTU
{PQT, PQRT} {PQU, PQRU} {QTU, QRTU}
PQRS QRST QRSU
{PQ, PQR} {QT, QRT} {QU, QRU} {RS, QRS}

Figure 4.3: Subproblems after QR is known

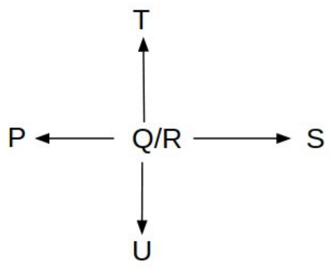


Figure 4.4: Join Graph after merging Q and R

For a join graph, the maximum length chain subgraph is called **backbone**. The edges which are not part of the backbone are called **ribs**. For the join graph in Fig. 4.5, blue edges form the backbone and red edges form the ribs. Star join graphs have two edges in the backbone and rest edges are ribs, whereas chain join graphs have all edges in the backbone and zero ribs. (Note that although the graph is directed, the largest chain component is with respect to the undirected version of the graph.)

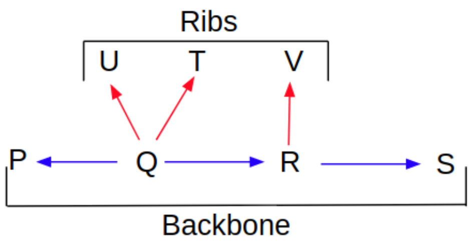


Figure 4.5: Join Graph showing backbone and ribs

Let a denote the number of ribs in a join graph. For a given number of edges as the number of ribs increases the graph becomes complex. We know that chain is a simple graph, it has zero ribs and star is a complex graph it has $D - 2$ ribs. The number of ribs can vary from 0 to

$TotalEdges - 2$ (from chain to star). $\mathcal{K}$ be a term defined as follows

$$\mathcal{K} = \begin{cases} a+2, & \text{backbone has more than 2 edges} \\ a+1, & \text{backbone has 2 edges} \end{cases} \tag{4.1}$$

Value of $\mathcal{K}$ varies from 2 to $D-1$. $\mathcal{K}$ also quantifies the closeness of branch join graph to star join graph. Let us assume that the backbone structure is fixed and we can move the ribs. Where should the ribs be placed to get the most complex graph possible w.r.t MSO? In our case, the complexity of a graph depends on the maximum number of revivals possible for completion of any dimension. As we have seen previously, it depends on the maximum number of adjacent edges of an edge, which occurs when all the ribs are attached on the same node and that node should be an internal node(degree greater than 1). e.g. among the join graphs in Fig. 4.6 the top one is complex, as it has the largest star subgraph possible using the ribs.

Lemma 2 For any join graph, the worst case sub-optimality occurs when all the ribs are attached to some internal node in the backbone.

In the remaining sections whenever we talk about branch join graphs, we mean the complex branch possible. In such complex join graphs, $\mathcal{K}$ is equal to the maximum degree of line graph for that join graph.

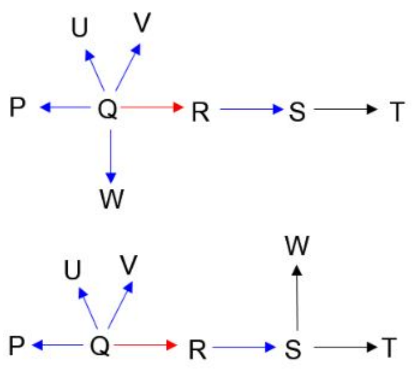


Figure 4.6: Distribution of ribs over same backbone

Now that we have quantified closeness of branch join graph, to that of star join graph using parameter $\mathcal{K}$, we will find the value of the maximum number of executions for any cost budget using it. For that, we will be using greedy edge picking. Greedy edge picking says that, to find the sequence of dimension completion leading up to the maximum number of executions, at each step we pick the dimension causing the maximum number of revivals. Using this we find the maximum number of executions for branch join graphs. Later in the appendix, we will prove that this greedy edge picking will indeed give the maximum number of revivals.

Theorem 1 *The maximum number of executions(terminated + complete) possible for any cost budget for a query with D dimensions is* $D + (D - \mathcal{K})\mathcal{K} + \frac{\mathcal{K}(\mathcal{K}-1)}{2}$

Proof:

We know that a terminated execution cannot reduce the dimensionality of the problem and a completed execution can revive the subproblems for which execution has been terminated with the same cost-budget. Hence, to increase the worst case number of executions we assume that the one subproblem which will complete will execute at the last. First time D dimensions are there, all will be executed but only the last one will complete.

After that, one completed execution leaves a (D -1) dimensional problem to be solved. The number of revivals by the completed subproblem will determine the number of executions next. We have already seen, it would depend on the number of neighbouring edges of the completed dimension.

For chain and star join graphs it would be 2 and $D - 1$ respectively. For any branch graph, it would be $\mathcal{K}$ in the worst case.

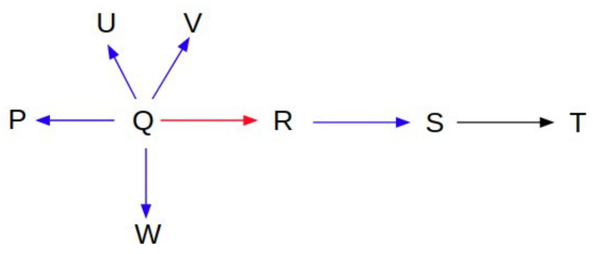
For any complex branch graph, the sequence of revivals in the worst case will depend on $\mathcal{K}$. It will be $\mathcal{K}$, while the number of edges remaining is more than $\mathcal{K}$. After that, the remaining $\mathcal{K}$ edges form a star structure, and thus have revival sequence as $\mathcal{K} - 1, \mathcal{K} - 2, .., 1$.

For chain, the sequence of revivals will be $\{2, 2, ..., 1\}$, because each edge has only two neighbouring edges. For star join graphs the sequence of revivals will be $\{D - 1, D - 2, ..., 1\}$, as all edges are neighbours of each other. (Sequence of revivals tells how many subproblems will be revived after completion of each dimension.)

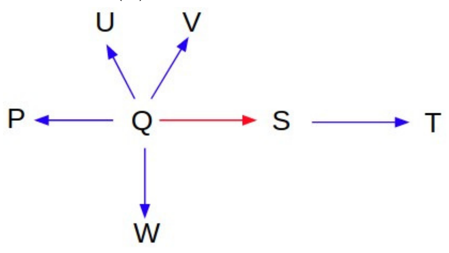
The order of revivals for a branch graph in the worst case is in such a way that the largest star subgraph is maintained until possible. Fig. 4.7 shows the order of dimension completion and revival for the given join graph. Red colour edges show the dimension completing and blue colour denotes the revived dimensions.

The maximum number of executions possible on any cost budget for any join graph will be $D + (D - \mathcal{K})\mathcal{K} + \{\mathcal{K} - 1 + ... + 1\}$, i.e. $D + (D - \mathcal{K})\mathcal{K} + \frac{\mathcal{K}(\mathcal{K}-1)}{2}$.

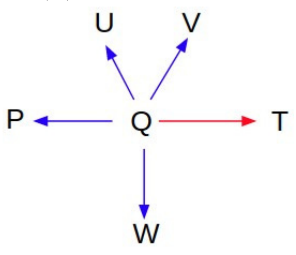
Now, that we have seen revivals, and how revivals affect the maximum number of executions per cost budget, can we say something about the execution sequence leading up to worst case sub-optimality? The next theorem talks about it, it says that for any join graph worst case sub-optimality occurs when the maximum number of executions happen on the last cost budget (which is the cost budget for which the query will complete).



(a) QR completes



(b) RS completes



(c) ST completes

Figure 4.7: Order of revivals in worst case for given join graph

Theorem 2 *Worst case sub-optimality for any execution occurs when all the revivals occur on the last cost budget*

Proof: Let's say the execution sequence for a query be $(D, D, ..., D, D + R)$ i.e., it says that for all cost budgets except the last one, we had D executions.

Where R denotes the maximum number of revivals for any cost budget.

The above execution sequence says that the worst case happens when all subproblems complete on last cost budget. Let C be the value of last cost budget, which is the k^{th} cost budget.

$Cost_1$: denotes cost of executing query using above sequence of executions. $Cost_1 = (D + R) * C + D * \frac{C}{2} + D * [\frac{C}{4} + ... + \frac{C}{2^{k-1}}]$

Let's say one of the dimension is learnt completely on $k - 1^{th}$ cost budget.

The new execution sequence will be $(D, D, ..., D + R', (D - 1) + R'')$ (where R = R' + R" – i.e. number of revivals on last cost budget).

$Cost_2$: cost of executing query using the new sequence. $Cost_2 = (D - 1 + R'') * C + (D + R') * \frac{C}{2} + D * [\frac{C}{4} + ... + \frac{C}{2^{k-1}}]$

$Cost_2 - Cost_1 = -C[1 + \frac{R'}{2}]$

So, even if one dimension completes before the last cost budget the cost of query execution decreases. Hence, worst case sub-optimality occurs when all revivals happen on the last cost budget.

MSO using this sequence will be, $\frac{Cost_1}{Copt}$. $Copt$ is the oracle's cost of executing this query, which is $\frac{C}{2} + \epsilon$ in this case.

$$MSO = \frac{(D + R) * C + D * \frac{C}{2} + D * [\frac{C}{4} + ... + \frac{C}{2^{k-1}}]}{\frac{C}{2}}$$

$$MSO = 2 * (D + R) + D * [1 + \frac{1}{2} + .. + \frac{1}{2^{k-2}}]$$

MSO can be upper bounded by

$$MSO \leq 2 * (D + R) + 2 * D$$

Note that, $(D + R)$ is nothing but the maximum number of executions for any cost budget. By replacing the value of $(D + R)$ with the value we proved in the previous theorem, we get general MSO as

$$MSO \leq 2 * D + 2 * (D + (D - \mathcal{K}) * \mathcal{K} + \frac{\mathcal{K}(\mathcal{K} - 1)}{2}$$

$$MSO \leq 4D + (2D - 1)\mathcal{K} - \mathcal{K}^2$$

As we know for star join graphs $\mathcal{K} = D - 1$, its MSO will be $D^2 + 3D$. Similarly, as $\mathcal{K} = 2$ for chain join graphs its MSO will be $8D - 6$.

Remember we used greedy edge picking to compute the maximum number of executions for any cost budget. Next we will show that this greedy order indeed gives us the maximum number of executions possible.

Greedy Edge Picking constructs the worst case sub-optimality. Previously we have discussed that in branch join graph we pick edges (which corresponds to dimension completion), in such a way that the number of revivals caused by that edge is maximum. Our claim was that this order of completion of edges will construct the worst case sub-optimality for the join graph.

From the previous discussions, two points must be noted down:

1. What is the effect of edge completion on a join graph

   When an edge is picked or in a way the dimension corresponding to that edge completes, vertices corresponding to that edge are merged. As shown in Fig 4.7.

2. To what structure does join graph converge due to by greedy edge picking

   The order of picking edges using greedy approach is in such a way that the join graph converges to the maximum star subgraph possible. E.g., in Fig 4.7 the maximum star subgraph possible consists of edges {QP, QU, QV, QR, QW} (five edges), this star structure is maintained in Fig 4.7c as well.

We have seen that worst case sub-optimality occurs when all the revivals happen on the last cost-budget. Hence, to show that greedy edge picking constructs the worst case sub-optimality, we only need to show that it causes the maximum number of revivals possible on any cost-budget.

Theorem 3 *Greedy edge picking causes the maximum number of revivals possible for any cost-budget*

Proof

Let's say initially the join graph is G.

Let the sequence of edges picked by greedy approach be $(e_1, e_2, e_3, ..., e_D)$.

Sequence of edges picked by any other approach be $(o_1, o_2, o_3, ..., o_D)$.

G_i denotes join graph after picking $(i-1)$ edges using greedy approach, and G'_i denotes join graph after picking $(i-1)$ edges using any other approach.

Let $f(e_i, G_i)$ denote the total number of revivals by picking edge e_i in graph G_i.

Total number of revivals by picking K edges is given by:

$$\sum_{i=1}^{D} f(e_i, G_i)$$

Initially, when we pick the first edge we know that

$$f(e_1, G_1) \geq f(o_1, G'_1)$$

because we pick the edge which causes maximum revivals.

Let us assume that after picking K edges:

$$\sum_{i=1}^{K} f(e_i, G_i) \geq \sum_{i=1}^{K} f(o_i, G'_i)$$

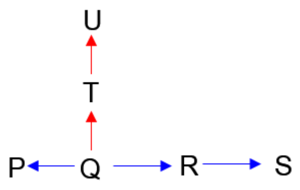
i.e., after picking K edges, the number of revivals caused by greedy approach is at least as good as any other approach.

Now, we only need to show that the number of revivals caused by $K+1^{th}$ edge picked using the greedy approach is more than or equal to that of the edge picked using any other approach.

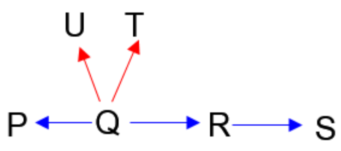
The remaining, $(D-K)$ edges could form a star, chain or branch structure. If they form a star structure, any edge picked will cause the same number of revivals (as each edge is connected to every other edge). If they form chain structure, apart from the first and last edge, rest all edges will cause 2 revivals and greedy approach will pick any edge causing 2 revivals. Now, coming to branch join graph case, as discussed earlier the greedy edge picking maintains the maximum star subgraph structure possible. We know that maximum revival occurs in case of star structure. Hence, in this case, also the edge picked using greedy approach will cause more (or equal, when the other approach also maintains the maximum star subgraph) revivals than edge picked using any other approach. As any other approach might not be able to retain the maximum star subgraph.

Hence, using induction hypothesis we can say that the total number of revivals caused by the greedy approach will be at least as good as any other approach.

Fig. 4.8 shows an example of best and worst case mid-way branch join graph. The best case happens when ribs also form a chain. Fig. 4.9 shows plot of MSO for star, chain and branch (mid-way complexity, i.e. $\mathcal{K} = 2 + \lceil \frac{D-3}{2} \rceil$, both best and worst case) join graphs with increase in number of edges.



(a) Best Mid-Way Branch



(b) Worst Mid-Way Branch

Figure 4.8: Best and worst case mid-way branch join graphs

There is an opportunity to make branch join graph MSO tighter, as we are considering worst case branch join graph. But actually, for a branch join graph depending on its structure, its MSO could lie anywhere between blue and yellow dot in Fig. 4.9. The actual tight MSO can also be computed based on the actual join graph, an algorithm for it is discussed late in this section.

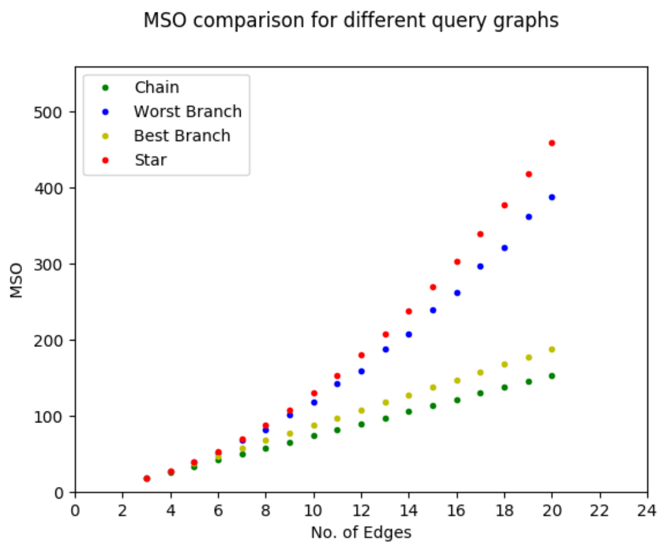


Figure 4.9: MSO plots for chain, star and mid-way branch join graphs

4.1 Change in MSO with $\mathcal{K}$

The following plot (Fig. 4.10) shows the change in MSO as the join graph changes from chain to star with a fixed number of edges. The left extreme on X-axis represents the chain graph, and right extreme represents star graph. Y-axis shows the MSO values.

Concavity of MSO equation The MSO equation gives the effect of complexity of join graph on MSO. For a given number of edges, when the graph is simple(i.e. chain) its MSO is $8D - 6$ and when it is complex(i.e. star) MSO goes to $D^2 + 3D$. For structurally intermediate branch graphs the MSO is bounded by the given formula.

The double derivative of MSO formula with respect to $\mathcal{K}$ is -2, hence the MSO curve for a fixed number of edges and changing $\mathcal{K}$ will be concave.

The slope for the MSO equation is given by $(2D-1) - 2\mathcal{K}$. As the slope is a linear equation of $\mathcal{K}$, there will be no steep increase or sudden jump in MSO as we move from chain to star.

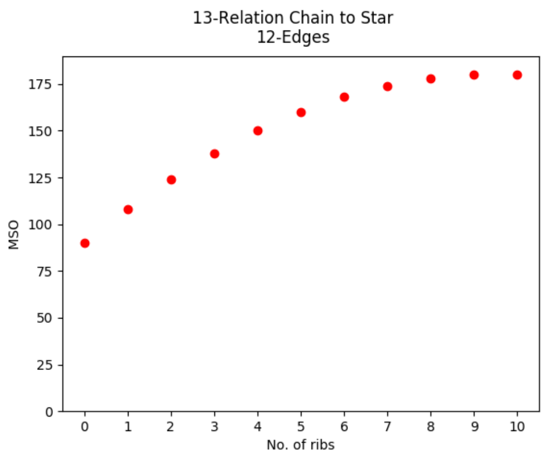


Figure 4.10: Plot showing change in MSO with change in graph structure from chain to star

In this plot, for branch join graphs we always considered the worst case, which is not always true (i.e. all ribs attached to the same node in backbone). In the next section, we give an algorithm which takes a join graph and gives its tight MSO bound.

4.2 Algorithm for MSO computation

The algorithm takes query graph as input. It forms the adjacency list from it and computes the degree of all vertices. Number of neighboring edges of a edge(v_1, v_2) is then given by, $Degree(v_1) + Degree(v_2) - 2$. Remove the edge with maximum neighbours from the graph, as it causes the maximum number of revivals. Adjust the adjacency list as if the end vertices of the removed edge are one vertex. Then follow the same process again, until only one dimension is left.

The general MSO formula discussed earlier will give MSO bounds same for join graphs in Fig. 4.8, i.e. equal to 40. But, it's not the same for them as the maximum number of executions for any cost budget is different for both of them (which is a function of how the ribs are attached). Algorithm 2 computes $(D + R)$, and then computes MSO using formula $2 * D + 2 * (D + R)$. Here, $(D + R)$ will be 14 and 15 respectively for join graphs Fig. 4.8a and Fig. 4.8b, their MSO will be 38 and 40. As the number of ribs increases the looseness of MSO bound for branch join graphs increases.

Algorithm 2 MSO Eval

```
1: Input: Query Graph
2: Output: MSO Value
3: $D$ : Number of error-prone edges
4: $R = 0$
5: $I = D$
6: while $I > 1$ do
7:     Compute degree of each vertex
8:     Find the edge with maximum number of neighboring edges
9:     $m = MaxNumberOfNeighboringEdge$
10:    Remove the edge corresponding to max $m$
11:    Merge the vertex corresponding to that edge
12:    $I = I - 1$
13:    $R = R + m$
14: end while
15: return $2 * D + 2 * (D + R)$
```

Query graph based MSO analysis can also be done for SpillBound, and similar MSO bounds can be achieved for SpillBound as well. This is because while deriving bounds for SpillBound, authors have considered that all dimensions will be revived, which is not the case always. Hence, SpillBound approach will never violate the query graph based MSO bounds achieved by Aqua.

Chapter 5

Implementation

Aqua execution works by executing the 1-D subproblem sets, for each cost budget and updating the minimum selectivity that is learnt. Notice that, the order of execution of these 1-D subproblems doesn't affect the MSO bounds. Following terms will be used in implementation :

1. **MinCard** : Minimum cardinality vector, has all 1's
2. **MaxCard** : Maximum cardinality vector, has maximum cardinality for each dimension
3. **RunCard** : Running cardinality vector, lower bound learnt on cardinality of each dimension
4. **ReachCard** : Reached Card vector, upper bound learnt on cardinality of each dimension
5. **Crun** : cost of optimal plan at RunCard
6. **Cmax** : cost of optimal plan at MaxCard

MinCard, MaxCard, RunCard and ReachCard are vectors of length equal to dimension. Initially, RunCard and ReachCard are equal to MinCard and MaxCard respectively, and $Crun = Cmin$. At each step to decide which dimension to execute, we find the plan at the intersection of diagonal between RunCard and ReachCard, and the cost budget, the dimension which is at the lower level in the plan is executed (this plan is referred as the diagonal plan). In Aqua, for each dimension (using intermediate query) we use CBP to identify the max SLP plan for the dimension for the given cost budget and then execute it. But, in the implementation we don't optimize the intermediate query for that dimension instead, we execute the sub-plan for the diagonal plan with the whole cost budget. This is done because, for implementing CBP and SLP to find the plan for an intermediate query, we will have to do more number of optimization calls, which comes at the cost of little or no gain in MSO. Because of this heuristic of not finding the optimal plan for the intermediate query for that dimension, we might suffer in MSO (discussed in next section).

Steps of implementations are:

1. Find the cost budget greater than Crun
2. Find the plan at intersection of diagonal and the cost budget
3. Pick the dimension at lower level in the plan
4. Execute the sub-plan for the dimension with full cost budget
5. Update RunCard with learnt cardinality
6. If sub-plan executes completely then Update ReachCard
7. Continue until all dimensions are known or $Cost(ReachCard)/Crun \leq 2$

Fig. 5.1 shows an example of ESS for a 2-D query (q_a denotes its actual selectivity location), which has two join predicates. x and y, correspond to the selectivity of join predicates. The dashed black arcs in the ESS correspond to the cost budgets. The dark green line in ESS shows the initial diagonal, which is drawn between $RunCard$ and $ReachCard$. Its intersection with the cost budget IC_1 (i.e. the selectivity location on the diagonal having optimal cost equal to IC_1) is shown in the red circle. Now, the plan at this intersection was picked, and the lower level subplan in it has x join predicate in it. This subplan is executed with budget IC_1, it doesn't terminate and learns some lower bound on x. $RunCard$ is updated, and a new diagonal is drawn between $RunCard$ and $ReachCard$ (Light green line). Now, as $Crun$ is more than IC_1, we move to cost budget IC_2. The execution continues according to the steps mentioned above.

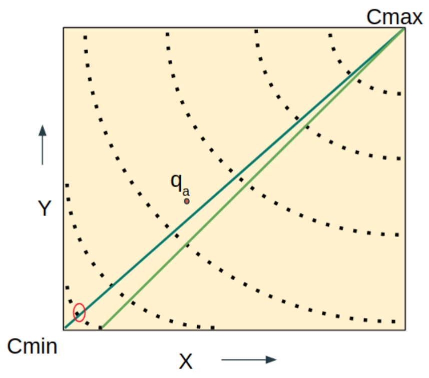


Figure 5.1: Example implementation of a 2-D query

Chapter 6

Experiments

In this chapter, we will compare the empirical performance of SpillBound and Aqua based on optimization calls made and MSO incured. SpillBound has all the information about the ESS pre-computed at compile time, whereas Aqua makes run-time optimization calls to gain similar information. As our implementation uses heuristics, we have to compare the two approaches empirically. For this, we take queries used in SpillBound paper and their reported MSO and ASO numbers. Queries are labelled as xD_Qy, where x denotes dimension of the query and y denotes query number. Queries 7, 26, 27, 91 and 96 are star-shaped, 15 is chain-shaped and 18, 19 and, 84 are branch shaped with $\mathcal{K}$ as 5, 4 and 3 respectively. The 3-D queries are executed at 100 resolution, 4-D queries on 30 resolution and other queries are executed at 10 resolution. Our algorithm is not resolution dependent, resolution in our case shows how many points in the ESS we evaluated the query for finding its sub-optimality.

Setup. We used PostgreSQL 9.4 with TPC-DS benchmark. All experiments are abstract cost-based.

Fig. 6.1, shows the MSO comparison between the two approaches. In almost all the cases the blue bar is below the red bar, except for in one or two cases (there also the difference is small).

Fig. 6.2, shows the average sub-optimality(ASO) comparison with SpillBound. For all the queries except $Q27$, ASO with Aqua is comparable to that of SpillBound. Even with the heuristic of not optimizing the intermediate query and just using the subplan on the diagonal, performance of Aqua is comparable to that of SpillBound. The few cases where Aqua is performing worse because of the heuristic. This heuristic performs badly when for the chosen dimension, the subplan chosen at the diagonal is not same as we would have got after optimizing the intermediate query for it (For Q27).

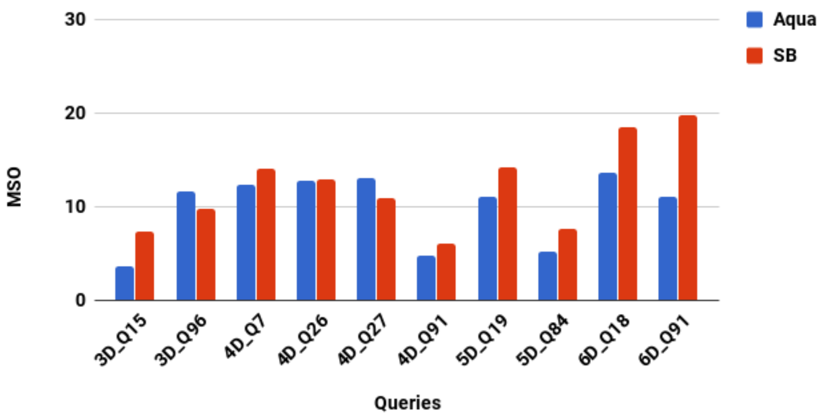


Figure 6.1: MSO comparison with SpillBound

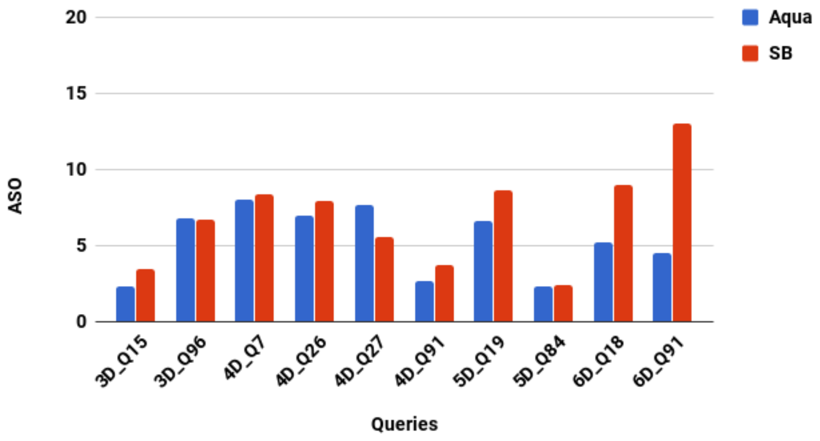


Figure 6.2: ASO comparison with SpillBound

We saw that the performance of Aqua is comparable to that of SpillBound. SpillBound achieves this performance by doing a lot of compile-time computation. Its compile time overhead is a function of resolution and number of dimensions. These overhead makes it impossible to be used with ad hoc queries and also as the dimension of queries increases, the compile time computations takes a huge amount of time. While Aqua achieves the same amount of performance, without any compile-time overheads and the number of optimizer calls made at run-time is also in few hundreds only (shown in Table. 6.1). Number of optimization calls can be upper bounded by $O(D * m * log_2 resolution)$.

Query	MaxOptCalls	AvgOptCalls
3D_Q15	93	64.2
3D_Q96	341	283.5
4D_Q7	327	270.6
4D_Q26	291	220.8
4D_Q27	387	315.3
4D_Q91	112	75.9
5D_Q19	345	243.3
5D_Q84	67	45.6
6D_Q18	243	116.1
6D_Q91	273	157.5

Table 6.1: Optimizer calls for given queries

At first these performance numbers look fishy, as with a subset of information (i.e. without constructing the ESS) of what SpillBound had, our MSO is comparable to them. This is because of the reasons discussed next. First, SpillBound contains a lot of information (whole ESS) for a query instance, all of it is not required. e.g., if the actual selectivity of query lies within k^{th} and $k+1^{th}$ cost budget, information about plans for cost budget beyond $k+1^{th}$ cost budget are not required. Second, there are some optimizations applied in our implementation which are not there in SpillBound, like, updating value of Cmax whenever one dimension is learnt. This saves us from making extra computations. Third, dependence of SpillBound algorithm on resolution. i.e., if the actual selectivity of a query is not aligned with ESS resolution then SpillBound will approximate the selectivity to the nearest point in the ESS and hence, it may incur some extra cost.

Due to low optimization calls per location in ESS and no compile-time overhead, Aqua has an edge over SpillBound approach. It is not only well suited for ad hoc query scenario but can also execute queries high dimensional queries. For canned queries, over a large number of executions, the run-time optimization overheads of Aqua will be more than that of compile-time optimization calls by SpillBound.

Chapter 7

Analysis without Selectivity Independence Assumption

Previously, we have seen and analyzed the algorithm for ad hoc query execution with selectivity assumption. Now, we will see what happens when we relax this assumption, which is also a cause for erroneous selectivity estimates. We will see that, as we relax this assumption dimension of a query could increase exponentially. Handling this increase in dimension is the main challenge here. Throughout the remaining sections, we will refer the case when selectivity independence assumption is relaxed as DEP case, and the case where this assumption is used as INDEP case.

Fig. 7.1 shows the cardinality expression of the subproblems when selectivity independence is not assumed for query Q1. Here, the selectivity of join predicate also depends on the join order. Note that, there are two ways of doing $P \bowtie L \bowtie S$, $(P \bowtie L) \bowtie S$ and $P \bowtie (S \bowtie L)$. Depending on the order of joins the selectivity of each join predicate varies. x_1 and y_2 are the selectivity of join predicate $P \bowtie L$ and $S \bowtie L$ respectively, when join predicate $P \bowtie L$ is applied first on L. x_2 denotes the selectivity of join predicate $P \bowtie L$, when it is applied after doing $S \bowtie L$.

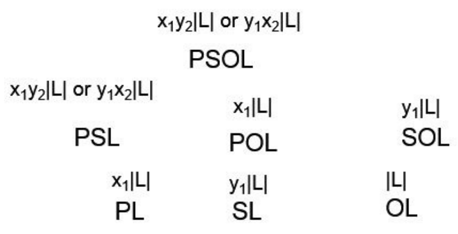


Figure 7.1: Cardinality expression for subproblems in Q1 in DEP case

Dimension of a problem in dependence case is defined as the number of subproblems with unique cardinality expression. PL and SL are in 1-D, whereas PSL is in 3-D. If we construct the ESS for PSL it will have two dimensions, x and y in selectivity independence space. Whereas without the selectivity independence assumption, since the join order has an effect on the selectivity of join predicates, there will be three dimensions x_1, y_1 and z (where $z = x_1y_2 = y_1x_2$). Here, the dimension z takes into account the join order factor for the dependence space.

For a star join graph with D join predicates the total number of subproblems will be $2^D - 1$. i.e., there is an exponential rise in dimension in this case when the selectivity independence assumption is relaxed. This is the challenge in the dependent analysis, now that the dimension has increased exponentially how do we handle it w.r.t. MSO?

A **Naive approach** assumes that, all subproblems are executed on all cost budgets, due to which this approach has an MSO of 4*$No\ of\ Subproblems$ [2]. Although we will see that this is not possible based on our approach to this problem.

A straightforward extension to the Indep algorithm will not work, because of the differences in Indep and Dep case described next. First, there are no revivals in DEP case, as each subproblem has a different dimension. Second, while in INDEP case we stop only when selectivities of all dimensions are known, in DEP case knowing selectivities of all dimensions is not required. Third, in INDEP case all dimensions are in active set initially, which is not possible in DEP case. Because of these differences, a straightforward extension of Algorithm 1 will not work for DEP case.

7.1 Algorithm for DEP Case Evaluation

Active Set consists of all the subproblems which are in 1D. Initially, all the level-1 subproblems are in 1D. e.g. in Fig. 2.2 subproblems PL and SL are in the active set. We only execute subproblems that are in the active set. Let's say PL completes at some cost budget, it will reduce the dimension of subproblem PSL. Similarly, completion of one 1D subproblem may reduce the dimension of some other subproblem. We refer to this as **Open up**, i.e. PL on completion opens up PSL.

e.g. let's say PL is in the active set and it completes on some k^{th} surface. From the cardinality expression given in Fig. 7.1, we can say that we know the value of x_1 as PL has completed. Thus, in one of the cardinality expression for PSL ($x_1y_2|L|$) we have just one unknown now y_2. Therefore, PSL now becomes a 1D subproblem and is added to the active set, and we say PL has opened up PSL.

For a given query, we start by finding the C_{min} and C_{max}. IC_k refers to iso-cost surface k and its cost is given by $\frac{C_{max}}{2^{m-k}}$ (where $m = log_2(\frac{C_{max}}{C_{min}})$). q_{lb} and q_{ub} are vectors having entry for selectivity of each dimension. D_e is the set of all subproblems in 1D. D^* denotes the dimension of the problem without independence assumption.

Algorithm 3 DEP Eval

```
1: $q_{lb} = (0, 0, ..., 0)$;
2: $q_{ub} = (1, 1, ..., 1)$;
3: $D_e = x_1, x_2, ..., x_D$
4: $k = 1$;
5: complete = false;
6: while complete == false do
7:     for $x_i$ in $D_e$ do
8:         $Q^{x_i}$ = findIntermediateQueries($x_i$);
9:         $P^k_{x_i} = CBP(Q^{x_i},\ x_i(q_{lb}),\ x_i(q_{ub}),\ cost(IC_k))$;
10:        execInfo = $CBE_{WB}(P^k_{x_i}, cost(IC_k))$;
11:        status = execInfo.status;
12:        $selec_{x_i}$ = execInfo.learnedSelec;
13:        if status == terminated then
14:            $q_{lb} = update(q_{lb}, selec_{x_i})$;
15:        else
16:            $q_{lb} = update(q_{lb}, selec_{x_i})$;
17:            $q_{ub} = update(q_{ub}, selec_{x_i})$;
18:            $D_e = D_e - x_i$;
19:            if $i == D^*$ then
20:                complete == true;
21:                break;
22:            else
23:                $D_i = D_i \bigcup OpenUp(x_i)$
24:            end if
25:        end if
26:    end for
27:    $k++$;
28: end while
29: execute $P_{opt}(q_{ub})$;
```

Algorithm 3 gives a way of executing a query without selectivity dependence assumption. It starts by initializing set D_e with active set. Then findIntermediateQueries gives the intermediate query for dimension x_i (i.e. gives the intermediate query for the subproblem corresponding to this dimension) and CBP gives the max SLP plan for it. CBP_{WB} executes the plan $P^k_{x_i}$ with the given cost budget IC_k and returns the execution information, which contains information about whether the plan executed successfully and what is the max SLP value. If the plan executes successfully then the selectivity of that dimension is known, q_{lb} and q_{ub} entry for that dimension are updated and it is removed from the D_e set. Dimensions corresponding to the subproblems opened up by it are added to set D_e using OpenUp function. If the plan doesn't complete then the q_{lb} for that dimension is updated. We stop when dimension D^*, which is the dimension corresponding to original problem completes. e.g. query Q1 has $D^* = 3$, $D = 2$. D^* corresponds to subproblem PSLO. Initially, {PL, SL, POL, SOL} are in active set, $q_{lb} = \{0, 0, 0\}$ and $q_{ub} = \{1, 1, 1\}$. We stop when dimension number D^* completes, which is 3 in this case.

Notice that, according to Algorithm 3, all subproblems can be executed on one cost budget only when all 1-D subproblem complete on the same cost budget and in turn opens up all the other subproblems, and opened up subproblems also complete on the same cost budget.

7.2 MSO Formula

We divide all the subproblems into three categories **Starting Set** are those subproblems which open up on first cost budget and complete on the first cost budget, **Repeating Set** subproblems open up on the initial cost budget and complete on the last cost budget (the last cost budget refers to the cost budget on which the original query e.g. PSLO completes) and **Finishing Set** subproblems open up and complete on the last cost budget. There are some other types of subproblems which might occur, like those which open up on intermediate cost budgets and finish up before or on the last cost budget. But for our analysis of the worst case, we only need to consider these three type of subproblems, although other types of subproblems can also occur but they will not be contributing much to the MSO. $\mathscr{S}$, $\mathscr{R}$ and $\mathscr{F}$ denotes the number of subproblems in Starting Set, Repeating Set and Finishing Set respectively.

Let's say that using our approach for the dependent case, the query PSLO completes on k^{th} cost budget having cost C. Then we can say that the cost of the optimal plan will be at least $\frac{C}{2}$. The MSO in terms of $\mathscr{S}$, $\mathscr{R}$ and $\mathscr{F}$ is given by

$$MSO = \frac{\mathscr{S}}{2^{k-2}} + 4 * \mathscr{R}(1 - \frac{1}{2^k}) + 2 * \mathscr{F} \tag{7.1}$$

Looking at the above equation we can say that, $\mathscr{S}$ has minimum impact on MSO, whereas $\mathscr{R}$ and $\mathscr{F}$ contribute more to the MSO. We are more concerned with the worst case MSO which occurs when k is large. As k becomes large the MSO equation is given by

$$MSO = 4 * \mathscr{R} + 2 * \mathscr{F} \tag{7.2}$$

So, in order to get the MSO, we should look to maximize Eq. 7.2.

7.3 Transformations to Increase MSO

In DEP case we start with all level-1 subproblems being in $\mathscr{R}$. When a subproblem in active set completes, it opens up some other subproblems. Using this fact we have to see, what kind of transformation (i.e. to consider which problems in the active set to move to $\mathscr{S}$ or to let them be in $\mathscr{R}$) should be applied which will increase the MSO.

Let's consider two transformations:

- **One subproblem opens up two new subproblems** - move one subproblem from $\mathscr{R}$ to $\mathscr{S}$, two subproblems move from $\mathscr{F}$ to $\mathscr{R}$. $\mathscr{R}$ count increases by one and $\mathscr{F}$ count decreases by two. From Eq. 7.2, we can see that this kind of transformation will not increase the MSO.

- **One subproblem opens up three new subproblem** - move one subproblem from $\mathscr{R}$ to $\mathscr{S}$, three subproblems move from $\mathscr{F}$ to $\mathscr{R}$. $\mathscr{R}$ count increases by two and $\mathscr{F}$ count decreases by three. From Eq. 7.2, we get that this increases MSO by 2.

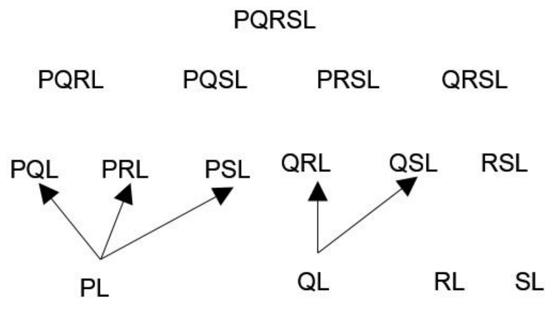


Figure 7.2: Both type of transformation being applied on the subproblems in active set

These are not the only transformations possible. There could be other transformations as well, but for sake of simplicity, they are discussed in the later section when they are required. e.g. one problem opens up two subproblems and one of them, in turn, opens up three subproblems, this type of transformation will also increase MSO. From the above discussion we can say that in order to get the MSO, we have to look at transformations where a subproblem in the active set opens up at least three new subproblems.

Chapter 8

Join Graph Specific Analysis without Independence Assumption

We will look at how the MSO will increase after relaxing the selectivity independence assumption. In DEP case each subproblem can be seen as a new dimension, whereas in INDEP case we have $n-1$ dimensions. Using the naive approach we have, $\mathscr{S} = 0$, $\mathscr{R} = Number\ of\ Subproblems$ and $\mathscr{F} = 0$. i.e. Assume that all subproblems are in $\mathscr{R}$.

According to algorithm 3, at any cost budget, we execute only the 1D subproblems. Initially, all the subproblems at level 1 are in 1D (which makes the active set). Completion of any of these subproblems will open up certain other subproblems. Using the above given two observations we will do the analysis for DEP case for two types of join graph.

8.1 Chain Join Graph Analysis

In this section we consider the queries having chain join graph. Fig. 8.1 shows an example of a chain join graph. A relation can have join predicates with atmost two relations. All the possible subproblems for the chain join graph in Fig. 8.1 are shown in Fig. 8.2. For any chain join graph query with n nodes, the number of subproblems is $\frac{n(n-1)}{2}$. A naive analysis in this case will give the MSO as $2 * D(D+1)\ or\ 2 * n(n-1)$.

$$\begin{tikzcd} P \arrow[r] & Q \arrow[r] & R \arrow[r] & S \arrow[r] & T \end{tikzcd}$$

Figure 8.1: Chain Join Graph for 5 relations

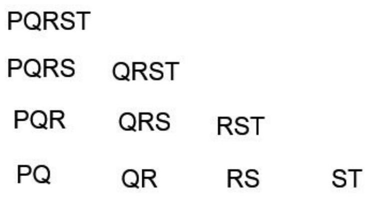


Figure 8.2: Subproblems for a chain graph with 5 relations

Initially we have, $\mathscr{S} = 0$, $\mathscr{R} = n - 1$ and $\mathscr{F} = \frac{(n-1)(n-2)}{2}$. The initial setting gives us some MSO value, can we apply some transformation to increase the MSO? As given in Section 7.3 a transformation can increase the MSO only if completion of one subproblem will open up atleast three subproblems. These kinds of transformation do not exist in case of chain join graphs, as one subproblem can open up atmost two subproblems.

Hence, the MSO occurs for chain join graphs at $\mathscr{S} = 0$, $\mathscr{R} = n - 1$ *or* D, $\mathscr{F} = \frac{(n-1)(n-2)}{2}$ *or* $\frac{D(D-1)}{2}$.

The MSO is given by

$$MSO = 4 * \mathscr{R} + 3 * \mathscr{F} = D^2 + 3 * D. \tag{8.1}$$

Notice that the MSO for Chain Join graphs remains same as for INDEP case, even after relaxing selectivity independence assumption. Table 8.1 shows a comparison between the MSO obtained for n relations using the naive approach, DEP case analysis, and INDEP case analysis.

n	Indep	Naive	Dep
4	18	24	18
5	28	40	28
6	40	60	40
7	54	84	54
8	70	112	70
9	88	144	88
10	108	180	108

Table 8.1: MSO comparison for chain join graph queries

8.2 Star Join Graph Analysis

Let's consider a query having n relations and has join graph of star type. The number of subproblems in this case will be $2^{n-1} - 1$ *or* $2^D - 1$. A naive analysis, in this case, will give the

MSO as

$$MSO = 2^{D+2} \tag{8.2}$$

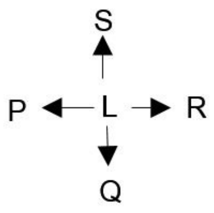


Figure 8.3: Star join graph for 5 relation

Let's consider the example of a star join graph of 5 relations (shown in Fig. 8.3). Initially, we have $\mathscr{S} = 0$ and $\mathscr{R} = D$. In this case certain transformations are possible which can increase the MSO.

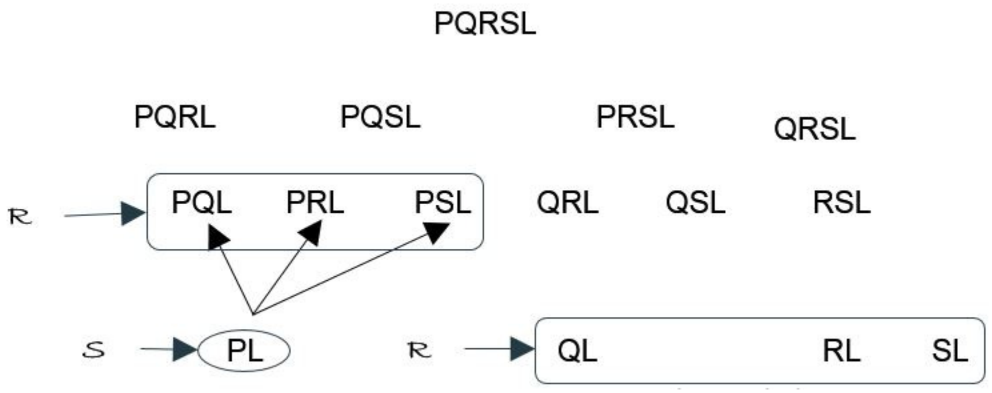


Figure 8.4: Worst case instance for star join of 5 relations

The active set consists of PL, QL, RL and SL. These set of subproblems are in 1D initially. We can see from the Fig. 8.4, that if PL completes it will open up three subproblems PQL, PRL and PSL. Fig. 8.4 shows the worst case instance for 5 relations star join. **Worst Case Instance Graph** shows the state of all the subproblems when MSO occurs for a given query. From it, we can depict the type of a subproblem in the following way, *Opening set* consists of nodes with outdegree atleast 1, *Repeating set* consists of level-1 nodes with outdegree zero and other level nodes with indegree one and outdegree zero and *Finishing set* consists of rest of the nodes.

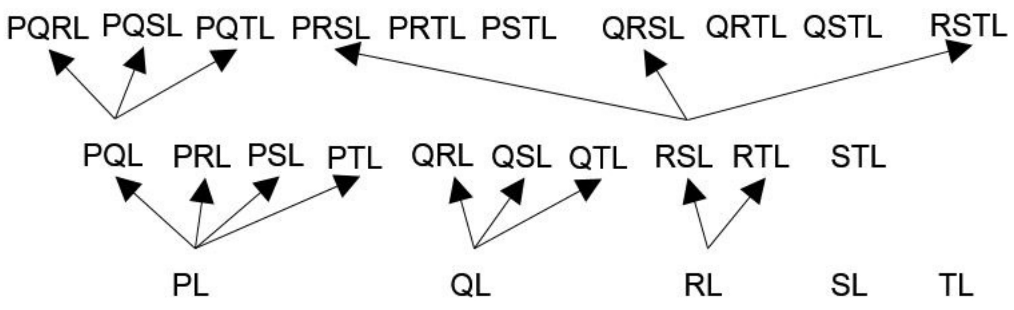


Figure 8.5: Worst case instance graph for star join of 6 relations

Section 7.3 shows the simplest kind of transformation that can be applied to increase the MSO. There are other transformations which can increase the MSO, one of which is shown in the Fig. 8.5. RL opens up RSL and RTL due to this transformation the MSO remains same. Then, RSL opens up PRSL, QRSL and RSTL. So, although the first transformation doesn't increase the MSO, but as RSL opens up three subproblems MSO increases by 2.

Just using the simple transformations given in section 7.3, we can give a lower bound on the worst case MSO for star join graphs, $MSO = 5 * 2^{D-1}$. For star join graphs the MSO is exponential in D, whereas for INDEP case it was quadratic in D. Table 8.2 shows a comparison of MSO in the three cases. The MSO bounds for DEP case given in the table 8.2 are found out empirically using a python script which uses a brute force approach. From the table 8.2 we can see that DEP gives 25% better guarantees than the naive approach.

n	Indep	Naive	Dep
4	18	28	20
5	28	60	40
6	40	124	82
7	54	252	172
8	70	508	352
9	88	1020	724
10	108	2044	1470

Table 8.2: MSO comparison for star join graph queries

8.3 Other Type of Join Graphs

Apart from the two join graphs analyzed in the previous sections, we can have other types of join graphs such as branch join graphs, cyclic join graphs, and complete join graphs. We did an analysis of join graphs for TPC-H and TPC-DS benchmark queries, for TPC-H queries we didn't find any query which has complete join graph, whereas for TPC-DS queries only three queries had complete join graphs.

Query	INDEP	NAIVE	DEP	Query Graph
H_Q5	40	60	40	chain(6)
H_Q7	40	60	40	chain(6)
DS_Q15	18	24	18	chain(4)
DS_Q96	18	28	20	star(4)
H_Q8	70	140	92	branch(8)
DS_Q7	28	60	40	star(5)
DS_Q26	28	60	40	star(5)
DS_Q91	54	160	106	branch(7)
H_Q7	40	60	40	chain(6)
DS_Q19	40	96	62	branch(6)

Table 8.3: MSO comparison for some TPC-H and TPC-DS queries

For a given number of relations n, the number of subproblems in branch join graph lies between that of chain join graph and star join graph. Hence, the MSO for branch join graph should lie in between that chain join graphs and star join graphs. Same applies to cycle join graphs, for them the MSO should be closer to that of the chain join graphs. Table 8.3 shows MSO values for some TPC-H and TPC-DS queries. For these set of queries, even after relaxing selectivity independence assumption, there is atmost 2 times increase in MSO bounds.

Chapter 9

Conclusions and Future Work

Aqua works by attacking the 1-D subproblems of a query. We have seen that the MSO depends on the type of join graph. Using parameter $\mathcal{K}$, a query graph dependent MSO formula was established. Empirical performance of Aqua approach is comparable to that of SpillBound, without any compile-time overhead and low run-time optimization calls. Hence, Aqua is well suited for ad hoc query execution. The same approach of attacking 1-D subproblems can be extended to work without selectivity independence assumption and give an algorithm for it. For chain join graphs the MSO doesn't change even after relaxing selectivity independence assumption, but for star join graphs it becomes exponential in the number of join predicates. MSO for other types of join graphs (apart from complete join graphs) should lie between that of chain join graphs and star join graphs. $\mathscr{R}$ type of subproblems contribute largely to the MSO. In our analysis we have considered that $\mathscr{R}$ type of subproblems will open up on initial cost budget and will complete only on the last cost budget, this is highly unlikely to happen in real execution of queries. Hence we believe that empirical evaluation of algorithm 3 should give us better bounds.

In future, we would like to replace the global diagonal approach for dimension picking with a local diagonal approach to maximize learning along each dimension. We also look to evaluate until how many executions of the same query template Aqua performs better than SpillBound approach. For DEP case, we would like to give a bound on the number of optimization calls, and also look to evaluate whether empirically DEP algorithm violates Indep bounds.

Bibliography

[1] A. Dutt. “Plan Bouquets: An Exploratory Approach to Robust Query Processing”. *Ph.D. Thesis, IISc Bangalore*, 2016. 2, 4, 10

[2] A. Dutt and J. Haritsa. “Plan Bouquets: Query Processing Without Selectivity Estimation”. *ACM SIGMOD*, 2014. ii, 2, 31

[3] S. Karthik, J. Haritsa, S. Kenkre, V. Pandit and L. Krishnan. “Platform-independent Robust Query Processing”. *IEEE ICDE*, 2017. 2, 3