Efficient and Robust generation of plans in Top-Down Optimizers

A Project Report
Submitted in partial fulfilment of the requirements for the Degree of Master of Engineering in Computer Science and Engineering

by
Rahul Sudhir Pandit

Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012

July 11, 2011
TO

My Parents

and

My GrandParents
Acknowledgements

I would like to thank my advisor Prof. Jayant Haritsa for his constant support and valuable guidance throughout my project work. He has always been a source of inspiration for me throughout my stay at IISc.

I would also like to thank Anshuman for his guidance in my project work. I would also like to thank my other lab mates for their help. I would also like to thank my friends for making the stay at IISc enjoyable.

I would also like to thank my parents and my grandmother for their constant support throughout two years of my stay at IISc. Lastly and most importantly, I would like to thank my sister for her constant encouragement and all her valuable advice given to me.
Abstract

Query optimization involves finding out the best plan for execution of a query among a set of plans. Modern database engines use the plan generated by cost based query optimizer to execute the query. Cost based query optimizers are basically divided into two groups based on the traversal of plans available in the plan space of the query. The two approaches are bottom-up query optimization and top-down query optimization. We present a study of top-down query optimization with respect to plan generation process and the quality of plans produced.

The plan space for any type of query is huge. Hence it is necessary to optimize the traversal of this space. We present an extensive analysis of the plan space traversal in top-down query optimizers for different types of queries and present a heuristic for improving the same.

Predicate selectivity at compile time could be significantly different from run time, leading to poor performance of compiler estimated optimal plans at run time. Expand[2] is an algorithm that generates plans that are more resistant to selectivity errors as compared to original plans. Expand[2] was implemented on bottom up optimizer. We have implemented Expand algorithm on top-down optimizer. We present an analysis of the plans generated by Pyro, a public domain top-down optimizer, armed with the Expand algorithm.
# Contents

Acknowledgements i

Abstract ii

1 Introduction 1
  1.1 Challenges ................................. 1
  1.2 Contribution ............................... 2

2 Related Work 3

3 Terminologies 5
  3.1 Operators .................................. 5
  3.2 MEMO, multiexpressions and groups ............. 6

4 Top-Down Optimizer Architecture 8
  4.1 Enumerator .................................. 9
  4.2 Plan Searcher ................................ 11

5 Efficient Traversal of Enumeration Lattice 14
  5.1 Motivation ................................ 14
  5.2 Problem Definition ......................... 17
  5.3 Heuristic Solution ......................... 18
    5.3.1 Good Seed .............................. 18
    5.3.2 Robust Traversal ....................... 19

6 Experiments for efficient traversal of enumeration lattice 20
  6.1 Experiments for finding out good seed ............. 21
  6.2 Experiments for robust traversal .................. 23

7 Generation of Robust Plans 27
  7.1 Problem Definition ........................... 27
    7.1.1 Plan Replacement ....................... 27
    7.1.2 Error Resistance Metrics ................ 28
    7.1.3 Problem Definition ...................... 29
  7.2 Robust Plan Generation ....................... 29
7.2.1 SEER algorithm ............................................. 29
7.2.2 Expand ......................................................... 30
7.3 Implementation Details ........................................ 31
  7.3.1 Foreign Plan Costing ...................................... 31
  7.3.2 Expanding the candidate set ............................. 32
  7.3.3 Selecting the candidate wagons ......................... 32

8 Experiments for Robust Plan Generation ....................... 33
  8.1 Plan Diagram Characteristics ............................... 34
  8.2 Plan Stability Characteristics ............................. 35
  8.3 Computation overheads .................................... 35

9 Conclusion .................................................... 37

A Appendix ....................................................... 39
  A.1 Query Templates ............................................ 39
  A.2 Type of query graphs ...................................... 41

References ....................................................... 44
List of Tables

5.1 Total Visits to the nodes in lattice ........................................ 16

6.1 Deterioration of the performance of heuristic over the best seed in no heuristic case ..................................................... 22

6.2 Improvement of the performance of heuristic over the average seed in no heuristic case ..................................................... 23

6.3 Robust traversal Heuristic ...................................................... 24

8.1 Plan Diagram Performance ................................................... 35

8.2 Plan Stability Performance ................................................... 35

8.3 Computational Overheads .................................................... 36
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Enumeration lattice of a query</td>
<td>7</td>
</tr>
<tr>
<td>4.1</td>
<td>Top-Down Optimization Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>4.2</td>
<td>Transformation rules in Pyro</td>
<td>13</td>
</tr>
<tr>
<td>5.1</td>
<td>Sample Query</td>
<td>16</td>
</tr>
<tr>
<td>6.1</td>
<td>Query</td>
<td>21</td>
</tr>
<tr>
<td>6.2</td>
<td>Plan generated with cost bound as 700</td>
<td>22</td>
</tr>
<tr>
<td>6.3</td>
<td>Plan generated by unconstrained optimizer</td>
<td>23</td>
</tr>
<tr>
<td>6.4</td>
<td>Traversal of enumeration lattice</td>
<td>25</td>
</tr>
<tr>
<td>6.5</td>
<td>Traversal of enumeration lattice</td>
<td>25</td>
</tr>
<tr>
<td>6.6</td>
<td>Traversal of enumeration lattice</td>
<td>26</td>
</tr>
<tr>
<td>8.1</td>
<td>Plan Diagrams</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Query optimizer is a very important component of the database engine. When a query is submitted to a database engine, the query optimizer first finds out an optimal plan to execute the query and then the engine uses that plan to execute the query. Traditionally the process of query optimization has been classified into two approaches, that of bottom-up query optimization and top-down query optimization. System R’s[11] bottom-up query optimizer architecture forms the basis of most commercial database engines today like DB2, Oracle and PostGreSQL among the more known ones. However this architecture had limited extensibility. For example adding a new operator like join or aggregation required a lot of changes in the optimizer. Graefe and DeWitt[8] proposed an architecture to generate plans in a top-down manner using transformations. Top-down approach has the added feature of extensibility. Commercial database engines like Microsoft use this top-down approach. Public domain optimizers like Pyro also use top-down approach for optimization. Pyro has been developed and maintained at IIT Mumbai. We have given a study of the top-down optimizers using Pyro.

1.1 Challenges

Top-down and bottom-up optimizers differ in the traversal of plan space. However both approaches have to explore same plan space for a query and face similar challenges with
respect to both plan generation process and quality of the plans produced. We restrict our study to the top-down optimizers for this report. If we consider the plan generation process, we face two challenges, one is to generate the plan space and other is to traverse the plan space to find out optimal plan with respect to cost. The plan space for a query is huge. For example in case of clique queries the number of feasible joins are of $\Omega(3^n)$, where $n$ is the number of relations in the query. Hence there is a need to optimize both the generation process and the traversal process.

We address another issue in our work. The predicate selectivity at compile time could be significantly different from that at run time, leading to poor run time performance. Such errors occur due to outdated statistics, attribute-value-independence assumptions and coarse summaries. Hence there is a need to generate plans that are more resistant to such errors.

1.2 Contribution

We provide an extensive analysis of the traversal process for a variety of queries. We change the traversal process by introducing a heuristic and thus optimizing the process for a subset of queries. We also provide an extensive analysis of the effect of our heuristic on the traversal process.

In order to analyze the plans generated by Pyro, we have ported PICASSO to Pyro. PICASSO[1] is a graphical visualizer tool that helps to analyze the Parametric Set of Optimal Plans(POSP)[9]. We have implemented an algorithm called Expand[2] to generate plans that are more resistant to selectivity errors.
Chapter 2

Related Work

Most of the commercial optimizers are built as variation of System R’s dynamic programming bottom-up approach. The bottom-up model was very elegant, however it lacked the feature of extensibility. Any new feature like a join or an aggregate operator when added to the optimizer required a lot of changes to the optimizer. With extensibility in view Graefe and DeWitt came up with Exodus[7], a transformations based top-down query optimizer. Graefe and McKenna came up with Volcano[8] with the goal of improving efficiency of top-down optimizers, through the concept of memoization. Shapiro et al.[12] provide a comparison of the performance of top-down and bottom-up optimizers. However the performance of the top-down optimizer in their analysis depended on pruning the cartesian products generated by the enumerator.

Any top-down optimizer has two main components, the enumerator and the plan searcher. The enumerator generates the plan space for a query and then the plan searcher searches for the most optimal plan with respect to costs of the plans in the plan space. The computational complexity of enumerating the plan space for a query was detailed by Ono and Lohman[10]. The plan space is exponential and in worst case is of the $\Omega(3^n)$ for clique queries. There has been lot of work in field of enumeration. DeHaan and Tompa[5] recently gave an algorithm that enumerates the plan space without cartesian products. However the algorithm used a complex data structure, but more importantly,
the performance of the algorithm degrades for queries that have clique join graph structure. Later on Moerkotte and Fender[6] came up with a very efficient, easy to implement algorithm to enumerate the plan space for any query graph. This is the best known algorithm for enumeration of join orders without cartesian products. The join reordering problem has also received significant attention which include traversal and enumerating all the plans through transformations. There also have been significant work on finding a near-optimal plan through greedy approach of traversal. Bruno, Cesar and Joshi[3] recently provided a polynomial heuristic to traverse the plan space greedily and end up with a near-optimal plan. None of the above address the challenges mentioned in the introduction. In order to find out optimal plan we need to traverse the complete plan space. While traversing the plan space, top-down optimizers carry out cost based sub-plan pruning, which is important for optimizing the process of traversal. To the best of our knowledge, there has been no work on optimizing the process of traversal.

Recently there has been a significant work over the quality of plans produced. SEER[4] is an algorithm that produces plans that are more resistant to selectivity errors from the plans present in POSP[9]. SEER treats optimizer as a black box and identifies alternate plans that are error resistant to selectivity errors from among the plans in POSP. Later on Expand[2] algorithm was proposed which generates alternate plans from complete plan space instead of just POSP. Expand being an intrusive approach requires changes in optimizer. However Expand has be implemented on bottom up optimizer only. There has been no analysis of the performance of Expand on top down optimizers.
Chapter 3

Terminologies

There are various terminologies used in context of top-down optimizers. However we would be following the terminologies that have been used by Shapiro et al.[12].

3.1 Operators

A logical operator is a function from operator’s inputs to its output. A physical operator is an algorithm mapping inputs to the output. For example join is a logical operator denoted by $\Join$, whereas nested loops join and merge sort join are examples of physical operators denoted by $\Join_N$ and $\Join_M$ respectively. An operator expression is a tree of operators in which children of an operator produce the operator’s inputs. An expression is logical or physical depending on whether the operators in the expression are logical or physical. A plan is an expression made up of physical operators entirely. Two operator expressions are logically equivalent if they produce identical results over any legal database. Plan space of a query is the complete set of physical operator expressions that are logically equivalent to the original query.
3.2 MEMO, multiexpressions and groups

The input of a query optimizer is an expression consisting entirely of logical operators along with a set of requested physical properties. Given the cost model, the aim of the optimizer is to find an optimal plan that is logically equivalent to the original query and has the requested set of physical properties.

Top-down optimizers since Volcano\cite{8} use a technique called memoization to find an optimal plan. The basic principle in memoization or even in bottom-up dynamic programming approach is the principle of optimality: every subplan of an optimal plan is itself optimal. Because of this principle, we only need to store the cost of the least costly subplan among all the logically equivalent subplans. MEMO is the compact representation of the plan space that was first presented in Volcano\cite{8} paper. A MEMO consist of two mutually recursive groups called groups and multiexpressions. A group is an equivalence class of expressions producing the same output. Following is a set of multiexpressions generated from the group of $A \bowtie B$.

$$A \bowtie B, A \bowtie N B, A \bowtie M B, B \bowtie A, B \bowtie N A, B \bowtie M A$$

A multiexpression is an operator having groups as input. A multiexpression represents all the expressions whose top operator is same and has the same inputs to the operator. A logical multiexpression has the top operator as the logical operator whereas physical multiexpression is the one that has top operator as the physical operator. During the process of enumeration each newly generated logical expression, corresponding to the group it belongs, is stored in the MEMO structure which avoids any duplicates being present in the final enumeration lattice.

Given a query, the enumerator first generates an enumeration lattice consisting of groups and multiexpressions. The top down optimizer traverses the lattice in a depth first search manner and finds out best plans for all the groups in the path. The best plans for each group is stored along with the group in the MEMO structure, thus ensuring that optimization of a group happens only once. Any further request for optimization of an
already optimized group can be addressed by MEMO structure itself.

(a) Expression tree consisting of groups

(b) Complete Expression tree for $A \times B \times C$

Figure 3.1: Enumeration lattice of a query
Chapter 4

Top-Down Optimizer Architecture

Top-down optimizer was introduced by Graefe and Dewitt through the Exodus\cite{7} optimizer which used transformation to enumerate the enumeration lattice. However Volcano\cite{8} architecture forms the basis of the top-down optimizers known today. In this section we describe the Volcano top-down optimizer architecture. We describe in brief the important components in the top-down optimizer.

Volcano top-down optimizer uses transformation to generate the enumeration lattice as against the traditional dynamic bottom-up optimizer which generates plans by building a lattice in a bottom up fashion starting with individual relations. Thus the two optimizers differ in traversal of the plan space. Top-down optimizer consist of enumer- ator and plan searcher as the two important components. In our experiments we have used Pyro optimizer and hence we have described components with respect to Pyro.

Pyro is public domain optimizer developed at IIT Mumbai. Pyro is based on Volcano\cite{8} architecture. Pyro is not a full strength optimizer as compared to any commercial optimizer. It has support for most of the basic functionalities of an optimizer. It takes the information of the statistics of the relations from a file and generates the cost of plan based on the cost model. It does not have any back-end engine and hence it only generates the best plan of query but has no functionality to execute the query. One of the major drawback of the optimizer is that it has no support for nested queries. However Pyro gives a good understanding of a top-down optimizer and we aim to study the
important components of top-down optimizer, with Pyro in view, in rest of this section.

4.1 Enumerator

Enumerator generates plan space of a query in form of an enumeration lattice. Enumerator gets the input in form of query. Enumerator extracts the information of the joins among the relations from query. This information is required to avoid cartesian products. Enumerator uses this information to generate the set of all the logical expressions that are logically equivalent to the input query.

Enumerator generates the enumeration lattice in form of groups and logical expressions. Figure 3.1(a) shows an enumeration lattice. The lattice consist of groups with topmost group being logically equivalent to the $A \Join B \Join C \Join D$. Each group consist of a list of logical expression that are logically equivalent. The inputs of topmost operator of each logical expression are shown with identically colored edges. Figure 3.1(b) shows a fine representation of the logical expressions in the group of $A \Join B \Join C$.

There have been various algorithms to generate the enumeration lattice. Traditionally transformations were used to generate the enumeration lattice. An important issue here was to avoid cartesian products in the lattice generated. DeHaan and Tompa[5] came up with an algorithm that generates the lattice, without cartesian products using a structure called biconnection tree. Moerkotte and Fender[6] came up with an even elegant and more efficient algorithm to generate the enumeration lattice without cartesian product. We have carried out our experiments on Pyro. The experiments depend on the enumeration process of Pyro and hence we would be describing the process in short in rest of the subsection.

**Pyro Enumerator** Pyro enumerator generates the enumeration lattice using transformation rules. Given a query the enumerator first generates a seed logical expression in form of left-deep tree. Then the enumerator applies the rules shown in Figure 4.2 to generate subsequent logical expressions. The associativity and commutativity rule provide a closed set of transformation rules to generate enumeration lattice. Pyro has added
Chapter 4. Top-Down Optimizer Architecture

**Optimize Query**

⇒ *Input*: Query Q  
⇒ *Output*: Optimal plan P for Q

1. Group G → Enumerate(Q)
2. Plan P → Optimize_Group(G,UB)

**Optimize Group**

⇒ *Input*: Group G of logical expressions, upper bound UB for cost and a set of physical properties T  
⇒ *Output*: Optimal plan P for G

1. if BestPlan[G] = NULL
2. Cost(BestPlan) = ∞
3. for all logical expressions Eᵢ ∈ G
4. Pᵢ → Optimize_Expression(Eᵢ,UB,T)
5. UB = min(UB,Cost(Pᵢ))
6. if Cost(BestPlan[G]) > Cost(Pᵢ)
8. return BestPlan[G]

**Optimize Expression**

⇒ *Input*: Logical Expression E, Upper Bound cost UB and a set of physical properties T  
⇒ *Output*: Optimal plan P for E

1. for all child group CGᵢ
2. BestPlan[CGᵢ] → Optimize_Group(CGᵢ,UB,Tᵢ)
3. if Cost(CGᵢ) < UB
4. UB = UB - Cost(CGᵢ)
5. else return NULL
6. find the cheapest plan P among all physical multiexpressions logically equivalent to E and satisfy T.
7. return P

Figure 4.1: Top-Down Optimization Algorithm
another transformation rule so as to speed up the process. The transformation rules are applied recursively to the logical expressions generated until no new logical expressions are generated.

Group consist a list of logical expressions. When a logical expression is generated, a check for uniqueness is made in the MEMO structure and if found out to be unique, the logical expression is added to the list of the corresponding group. An entry is made in the MEMO structure as well. Thus MEMO structure avoids presence of duplicates in the enumeration lattice. An observation that could be made here is that the sequence of the logical expressions in the list of a group depends on the seed logical expression.

4.2 Plan Searcher

In the previous subsection we described the process of generating the enumeration lattice. Now that we have generated the enumeration lattice, the next step is to traverse the lattice to find out the best plan for the query. Plan searcher traverses the enumeration lattice in a depth first search manner. The optimized groups have their optimal plans stored in MEMO structure. This ensures that all the groups are optimized only once. Figure 4.1 shows the basic algorithm for Plan Searcher.

After the enumeration lattice is generated the topmost group in the lattice is given as an input to Optimize_Group function along with upper bound cost as $\infty$ and a set of physical properties. Each group consist of logically equivalent logical expressions in form of a list. Each logical expression in the group is traversed to find out a plan to execute the expression. The cost of the plan for executing the logical expression serves as the upper bound cost for the remaining unoptimized logical expressions in the group. Line 5 in the Optimize_Group function sets the upper bound cost for the group after each expression has been traversed. Upper bound cost is sent as input to the Optimize_Expression function along with the logical expression. If the plan returned from this function has lower cost than the current best plan for the group, then that plan is stored as the best plan for the group.
OptimizeExpression function takes input as the expression, the upper bound cost and a set of physical properties. The topmost operator of the logical expression has inputs as groups. The first step in this function is to optimize the input groups. The upper bound is reset after the best plan for each group has been found. Once the best plans for all of the input groups of the logical expression are found out, we evaluate the best plan for the current logical expression. In this step we apply the different physical operators on the inputs. Thus we explore the space of all the physical multiexpressions that are logically equivalent to the logical expression. Thus the plan generated by the multiexpression that satisfies the required physical properties and is best among the logically equivalent physical multiexpressions is returned by this function.

**Pruning** Top-down optimizers potentially have the ability to prune subplans of current plan if current plan is already violating the upper bound. In line 3 of OptimizeExpression algorithm, a check is made to find out if the upper bound is violated. If the upper bound is violated the current logical expression is not evaluated further. Thus the remaining inputs are not traversed. The space of multiexpressions is also not explored. This leads to reduced computations during traversal.

**Group-Pruning** Top-down optimizers prune groups while traversing a logical expression. However that group could be traversed through different logical expressions. For example, in the enumeration lattice in Figure 3.1(a) the group $A \Join B$ could be traversed while traversing the logical expressions $(A \Join B) \Join (C \Join D)$ and $(A \Join B) \Join C$. Thus even if the group is pruned while traversing one logical expression it’s not necessary that it would be pruned in others. Thus a significant pruning case would be where the group is pruned during the traversal of all the logical expression that contained it and thus is not traversed at all. This we define as group-pruning. Group-pruning is important because it implies that top-down optimizer does better than bottom-up optimizer during traversal. Top-down optimizer is able to prune a group which bottom-up optimizer has to optimize.
Figure 4.2: Transformation rules in Pyro
Chapter 5

Efficient Traversal of Enumeration Lattice

In the previous section we gave an overview of the two important components of top-down optimizers. The best algorithm for generating the enumeration lattice was recently proposed by Moerkotte and Fender[6]. Now that we have the best known algorithm for generation of enumeration lattice, the next step is to optimize the traversal of the lattice. In this section we describe our contribution towards optimizing the process of traversal. First we describe the motivational scenario for our work, then we describe the problem definition and finally a heuristic solution to the problem.

5.1 Motivation

The join operations in an enumeration lattice is exponential with the number of relations in the query and is of $\Omega(3^n)$ in case of clique queries. Hence it is important to optimize the process of traversal. An important aspect of top-down optimizer is pruning. Our aim is to achieve maximum possible pruning. We have also introduced the concept of group-pruning in Section 4. A very intuitive way to achieve pruning is to obtain near optimal upper bound cost for each group early on. If we obtain the optimal plan from the first logical expression present in each group we would be able to achieve the maximum
Observation The costing of the top operator of logical expressions vary according to the height of the group to which the logical expression belongs in the enumeration lattice. As we go up the lattice the costing of the operators goes on decreasing. This is due to the fact that the size of the inputs is highest at the bottom-most level of the enumeration lattice. As we go up the lattice the size of the inputs keep on decreasing resulting in lower operator costs. While traversing in a depth first search manner, the operators at higher levels do not reduce the cost bounds by a large amount. Group-pruning could be achieved if in a logical expression one of the child group exceeds the upper bound cost of the logical expression. If the cost of one of the child group exceeds the upper bound cost then we can prune the other child groups. However intuitively such cases would occur in the lower part of the lattice. Hence we need to traverse to lower levels in the enumeration lattice to observe actual pruning of groups. An immediate conclusion that we can make from the observation is that even if we are able to achieve a near optimal cost early on in the traversal, still we wont be able to achieve pruning of a big group.

In order to verify the hypothesis based on above observation we studied group-pruning over various queries. Modified TPCH queries were used for the study over TPCH database. No group-pruning was observed in the study. Hence we now focus at another issue. During traversal a logical expression could be traversed multiple times. We want to minimize the total visits to the logical expressions present in the lattice. In the rest of the section we use the word ‘node’ to denote logical expression.

A node could be visited multiple times when the node is not optimized during traversal of one of the logical expressions containing it. Following are the causes for multiple visits to a node:

1. One of the child group of the node exceeds the upper bound cost limit, resulting in returning from the process of optimizing of the node without an optimal plan.

2. The node does not satisfy the physical properties required by parent logical expression.
Chapter 5. Efficient Traversal of Enumeration Lattice

<table>
<thead>
<tr>
<th>Type of Query</th>
<th>Visited Nodes</th>
<th>Total Visits to Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>26</td>
<td>100</td>
</tr>
<tr>
<td>Cycle</td>
<td>46</td>
<td>173</td>
</tr>
<tr>
<td>Star</td>
<td>87</td>
<td>429</td>
</tr>
<tr>
<td>Clique</td>
<td>57</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td>82</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2126</td>
</tr>
<tr>
<td></td>
<td>2287</td>
<td>28210</td>
</tr>
</tbody>
</table>

Table 5.1: Total Visits to the nodes in lattice

We show the analysis for the number of nodes that are visited in the enumeration lattice for a query against the total visits to the nodes present in the lattice. Table 5.1 shows the comparison of the two for different types of queries.

![Sample Query](a) ![Sample Query](b)

Figure 5.1: Sample Query

As we can see from the Table 5.1 the total visits to the nodes in the lattice is much larger than the nodes that are visited in the lattice. The total visits to the nodes in an enumeration lattice depends on upper bound cost set on a group. Hence it would depend on the sequence in which the nodes present in a group are traversed. We aim to find out a traversal that minimizes the total visits to the nodes present in enumeration lattice.

We have described the Pyro enumeration process in Section 4. Pyro follows the method of transformation to generate the enumeration lattice. Hence the sequence in which the logical expressions are traversed depend on the seed provided to the enumerator. If we provide different seeds we get different traversals of the enumeration lattice.
We performed an experiment where we provided all possible *seeds* to the enumerator to analyze the performance of the optimizer with respect to the different traversals. The permutation of sequence of the relations in the from clause would generate all possible left-deep *seeds* for our experiment.

Figure 5.1(a) shows a histogram representation for our experiment. We have used a star query consisting of 7 relations. Hence 7! *seeds* were sent to the enumerator in our experiment. Each *seed* would correspond to a unique way of traversal of the enumeration lattice. The histogram shows the distribution of the number of *seeds* against the total visits to the nodes in enumeration lattice. X axis consist of buckets of length 10, and denotes the total visits to the nodes in the enumeration lattice. Y axis denotes the number of *seeds*. A bar of height 110 at the point 2230 on X axis indicates that 110 *seeds* out of the 7! *seeds* have the total visits to the nodes in the range of 2230 and 2240. A cumulative description of the data is shown in Figure 5.1(b). A point (2230,1317) on the cumulative graph denotes that 1317 *seeds* out of 7! *seeds* have total visits to the nodes below 2230.

Figure 5.1(a) shows the range of the total visits to the nodes in the enumeration lattice for Star Query(c). The range extends from 1890 to 3450. Thus the best case performance is of an order half of the worst case performance. From above we can conclude that an optimal traversal can improve the performance of traversal by a factor of 2 for the Star Query(c). One more point to note here is that the *seeds* used in our experiments are left-deep *seeds*. Hence we are studying a subset of the traversals possible for the query. For example, a right-deep *seed* logical expression would generate a traversal that we have not considered in our experiment. Thus the range could go in either direction. In any case, the worst case performance would be at least twice the best case performance.

### 5.2 Problem Definition

Let the set of the unique ways in which an enumeration lattice could be traversed be called as the traversal space of the enumeration lattice. Each unique way in which the
lattice is traversed be called as traversal.

**Problem** Given an enumeration lattice, find out a traversal from the traversal space of the lattice which is optimal in terms of the total visits to the nodes in the enumeration lattice.

### 5.3 Heuristic Solution

We have given the problem definition in previous subsection. In this subsection we provide a heuristic solution to the problem. We attack the problem on two fronts:

1. To find out a good *seed* for given query.
2. Irrespective of the *seed* provide robust traversal.

In rest of the section we describe the heuristics to solve the problem on the two fronts.

#### 5.3.1 Good Seed

A good *seed* is the one that gives the total visits to the nodes near optimal. A good *seed* for the Star Query(c), whose histogram is shown is Figure 5.1(a), is the *seed* that has total visits to the nodes near the value of 1890. In our process to find out a good *seed* we considered external characteristics like the total number of joins on a relation (degree of the relation), the indexes on the attributes of the relations that participate in joins and the size of relations.

**Observation** We have showed the analysis of a star query for our motivational scenario. A star query has a central relation and the rest of the relations form a join with the central relation thus forming a star like structure. The good *seeds* for the scenario were the ones where the central relation was the left child of the bottommost operator. Hence degree of relations could serve as good heuristic for finding out good *seeds*.

The number of indexes on a relation also serve as good heuristic for finding out good *seeds*. From our observation over several queries we came to a conclusion that the relation
having the highest number of indexes participating in joins should be the left child of
the bottommost operator.

A further observation that was made in both of the cases above was that the relation
with second highest degree or the relation having second highest number of indexes
participating in joins should be the right child of the topmost operator. Thus the relations
having the highest and second highest characteristics mentioned above should be the ones
who are at the extreme ends in the from clause.

5.3.2 Robust Traversal

We gave a heuristic solution for providing a good seed in previous subsection. In this
subsection we provide a heuristic that gives robust performance irrespective of the seed
for enumeration. In this scenario we define robustness with respect to the total visits
to the nodes in the lattice. We aim to improve performance of bad seeds, meanwhile
compromising the performance of good seeds by a small factor. We also aim to improve
the average case total visits to the nodes in the lattice.

The sequence of traversal of logical expressions in a group determines the upper bound
cost that is established on the group. A near optimal upper bound cost being established
early on for a group is desirable. Hence the sequence of traversal of the logical expressions
in a group is also important for optimizing the total visits to the nodes in lattice. In
order to give sequence to the logical expressions in a group, we consider characteristics
like the height of the logical expression and the selectivity of the logical expression.

Selectivity The size of output of logical expression serves as an important characteris-
tic for sequencing. However the size of output of all the logical expressions in a
group is same. Hence we look at the size of the inputs of the topmost operator of the
logical expressions. Selectivity of an operator is a function of size of the inputs. In each
group we sort the logical expressions in descending order of the selectivities. We give an
analysis of this heuristic in the next section.
Chapter 6

Experiments for efficient traversal of enumeration lattice

In this section we give an analysis of the heuristics suggested in the previous section. We attacked the problem on two aspects. Both the aspect vary from each other significantly. One aspect considers the characteristics of the relations in the query while the other considers internal characteristics of the logical expressions in the enumeration lattice. Hence we differentiate the experiments for the two heuristics.

We have used the Pyro top down optimizer for our experiments. Pyro optimizer has two options for optimization. The two options being constrained top down optimization and the other being unconstrained top down optimization. The constrained top down optimization has cost based pruning enabled whereas the unconstrained optimization has cost based pruning disabled. For the purpose of our experiment we have used constrained optimization. We have performed the experiments on TPCH database with indexes on the attributes that are present by default in TPCH database.

There was some inconsistency observed in the behavior of the Pyro top-down optimizer for the constrained optimization case. The optimal plans generated for constrained and unconstrained optimization were different. The Figures 6.2 and 6.3 shows an example for the different plans generated by constrained and unconstrained optimization for the query in Figure 6.1. In order to reproduce the bug mentioned, use TPCH database
with index present on all the attributes of all the relations. However the optimizer is relatively stable for high cost bounds, in the sense that they are near optimal. In our experiment we have taken care that we provide a very high cost bound.

6.1 Experiments for finding out good seed

Here we have to verify two heuristics. The first heuristic is about the degree of the relations in the query and the second heuristic is based upon the indexes on the attributes of the relations present in the query. First we will describe the environment for the two heuristics. In the first case we verify the heuristic for degree of the relations in the query. For this case we generated queries that had relations of varying degree in the join graph. There were no indexes built on any relation in the database. In second case we verify the effect of indexes on performance of query. We used clique queries for this case. Indexes were built on relations in a manner to provide varying indexes for the relations participating in the query.

Table 6.1 shows the performance of the two heuristic over different randomly generated queries. The average case performance is defined as the percentage deterioration over the total visits to the nodes in lattice averaged over several queries. We always compare the performance of our heuristic against the optimal left-deep seed. Hence the value of average case performance would range from $[0, \infty)$, 0 being the desirable value. The best case performance indicates the best performance among all the experimental queries and similarly the worst performance indicates the worst performance among all
Figure 6.2: Plan generated with cost bound as 700

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>% Best Performance</th>
<th>% Worst Performance</th>
<th>% Average Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree</td>
<td>3.39</td>
<td>23.57</td>
<td>10.07</td>
</tr>
<tr>
<td>Index</td>
<td>1.25</td>
<td>21.29</td>
<td>13.20</td>
</tr>
</tbody>
</table>

Table 6.1: Deterioration of the performance of heuristic over the best seed in no heuristic case.

The queries used in the experiment.

From the table we can conclude that we are able to achieve near optimal performance. Table 6.2 shows the improvement of the heuristic over the average seed in no heuristic case. Here the values would indicate improvement as compared to previous case which indicated deterioration. We achieve good performance for both the cases. However we were not able to combine the two heuristics to form a generic heuristic. Generally queries are a mixture of the two cases that we considered. In other words queries have relations varying in both degree and indexes present simultaneously. Hence we need to find a way combine the effect of the two heuristics. Also one more point to note here is that we are considering optimal performance as the one having the best performance among the
Chapter 6. Experiments for efficient traversal of enumeration lattice

Figure 6.3: Plan generated by unconstrained optimizer

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Best Performance</th>
<th>Worst Performance</th>
<th>Average Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree</td>
<td>21.23</td>
<td>-3.22</td>
<td>12.20</td>
</tr>
<tr>
<td>Index</td>
<td>17.83</td>
<td>-6.22</td>
<td>9.50</td>
</tr>
</tbody>
</table>

Table 6.2: Improvement of the performance of heuristic over the average seed in no heuristic case

left-deep seeds. Hence the performance could be optimized even further.

6.2 Experiments for robust traversal

Pyro uses a left-deep seed for enumeration. Hence even if we find out the best seed for a query, we would obtain a traversal that is optimal among the left-deep seeds. However there could be a traversal that could perform better than this seed. Another problem with the heuristic described in previous section was that it was very restrictive. The heuristics for the two cases mentioned in previous section could not be merged to form a single heuristic for general queries.
Chapter 6. Experiments for efficient traversal of enumeration lattice

<table>
<thead>
<tr>
<th>Type of Query</th>
<th>% Best Performance</th>
<th>% Worst Performance</th>
<th>% Average Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>-13.20</td>
<td>21.02</td>
<td>1.27</td>
</tr>
<tr>
<td>Cycle</td>
<td>-1.44</td>
<td>7.22</td>
<td>1.29</td>
</tr>
<tr>
<td>Star</td>
<td>14.20</td>
<td>29.13</td>
<td>9.87</td>
</tr>
<tr>
<td>Clique</td>
<td>2.6</td>
<td>7.85</td>
<td>6.3</td>
</tr>
</tbody>
</table>

Table 6.3: Robust traversal Heuristic

We aim to provide a heuristic that could be applied over different types of queries. We alter the traversal of enumeration lattice internally by sequencing the logical expressions in a *group*. We arrange the logical expressions in a *group* in decreasing order of their selectivities. For the experiments related to current heuristic we use the four standard types of queries 1) Chain query 2) Cyclic query 3) Star query and 4) Clique query. More information of these query types is given in Appendix.

We have done an extensive analysis of performance of the heuristic over the four types of queries. We provide all possible left-deep *seeds* as an input to Pyro in both heuristic and non heuristic case and collect the data in form of total visits to nodes. We represent the data thus collected in form of histograms. The heuristic is represented by blue color histogram whereas the red color histogram shows the no heuristic case. The explanation of the histogram is given in Section 5. The queries corresponding to the histograms are given in the Appendix.

Figure 6.4(a) shows the histogram for a chain query. The heuristic reduces the variance of the histogram. It provides improvement in worst case performance. However there is a compromise both in best case performance and average case performance. Figure 6.4(b) shows the histogram of a cyclic query. The histogram of cyclic query also shows a reduction in variance. From the figure we can see that we get good best case, worst case and average case performance. However the heuristic shows a mixed effect for other cyclic queries. There are cyclic queries where the average case performance deteriorates.

Figures 6.5(a), 6.5(b) and 6.6(a) show the performance of star queries. We get the best performance for the heuristic in case of star queries. Figures 6.5(a) and 6.5(b) show
a great reduction in variance of the histogram. Not only the range over which the total visits to the nodes is reduced, but the range is towards the lower part of the range over which the no heuristic case extends. It is evident from figure that we get good best, average and worst case performance.

Figure 6.4: Traversal of enumeration lattice

Figure 6.6(b) shows the performance of a clique query. From the figure we can see that the range of the histogram in case of heuristic crosses the range of values in histogram of no heuristic case. We achieve performance improvement even over the best seed in no heuristic case. This example shows that we can achieve traversal which is not achieved by the optimal left-deep seeds, through the selectivity heuristic. It is evident from the diagram itself that we achieve good best case, average case and worst case performance for the query.

Figure 6.5: Traversal of enumeration lattice
Table 6.3 shows the performance of selectivity heuristic averaged out over unique randomly generated queries of each type. Best performance shows the percentage improvement of the best traversal in case of heuristic over the best traversal in case of no heuristic case averaged out over the test queries. Similarly we define the worst case and average case performance. From the tables it is clear that the heuristic works very well for clique and star queries. The positive values of best case performance indicates that the heuristic can achieve performance better than the optimal left-deep *seed* for clique and star queries. We get a small improvement in average case performance for chain and cyclic queries.
Chapter 7

Generation of Robust Plans

The plans generated by optimizer could perform poorly at run time because of predicate selectivity errors. We aim to generate plans that are relatively less sensitive to selectivity errors. More specifically, during compile time we want to identify plans that are near-optimal in absence of selectivity errors and comparatively stable in presence of selectivity errors. We identify these plans as “robust” plans. If the original plans are robust they are retained. We aim to find out plans that are marginally expensive locally but provide good global performance.

7.1 Problem Definition

7.1.1 Plan Replacement

Given a query point $q_e$ with the compiler estimated optimal plan $P_{oe}$, the run time location of query point could be at $q_a$ with optimal plan as $P_{oa}$. Let the replacement plan be denoted by $P_{re}$. The cost of a plan $P$ at point $q$ is denoted as $c(P,q)$. The point $q_a$ could be present in any one of the following regions of $P_{re}$:

1. **Endo-optimal region:** This is the region where $P_{re}$ is optimal. Hence $c(P_{re},q_a)$
   \[ \leq c(P_{oe},q_a), \]
   implying the replacement provides improved resistance to selectivity errors,
2. **λ-optimal region**: In this region the replacement plan is within λ bound of the optimal plan. In other words $c(P_{re},q_a) \leq (1 + \lambda) c(P_{oa},q_a)$. Now there are two possibilities. If $c(P_{re},q_a) < c(P_{oe},q_a)$, then the replacement guarantees resistance to selectivity errors. If $c(P_{re},q_a) > c(P_{oe},q_a)$, still we are assured that the harm is within bound.

3. **Exo-optimal region**: Here there are no guarantees of the performance of the replacement plan $P_{re}$.

The *endo-optimal* and the *λ-optimal* region are collectively called as *safe-regions*.

### 7.1.2 Error Resistance Metrics

Stability has been defined as the extent by which the replacement plan $P_{re}$ bridges the gap between $P_{oe}$ and $P_{oa}$ at $q_a$. The stability of plan replacement has been quantified based on *Selectivity Estimate Resistance Factor* (SERF) metric defined in [4].

\[
SERF(q_e, q_a) = 1 - \frac{c(P_{re}, q_a) - c(P_{oa}, q_a)}{c(P_{oe}, q_a) - c(P_{oa}, q_a)}
\]

SERF captures the fraction of the performance gap between $P_{oe}$ and $P_{oa}$ at $q_a$ that is covered by $P_{re}$. In principle SERF values can range over $(-\infty, 1]$. SERF values between $(0,1]$ indicates the replacement is beneficial, whereas the values between $[-\lambda,0]$ indicates the replacement neither hurts nor harms. However the SERF values between $(-\infty,-\lambda]$ indicates harmful replacement.

In order to capture the impact of replacement over the complete selectivity space $\text{AggSERF}$ has been defined.

\[
\text{AggSERF} = \frac{\sum_{q_e \in \text{rep}(S)} \sum_{q_a \in \text{exooe}(S)} \text{SERF}(q_e, q_a)}{\sum_{q_e \in (S)} \sum_{q_a \in \text{exoe}(S)} 1}
\]
7.1.3 Problem Definition

Given a query $q_e$ in selectivity space $S$, robust plan selection problem is defined as follows. The replacement plan should obey the notion of stability by obeying following rules:

1. **Local near optimality** $\frac{c(P_{re}, q_e)}{c(P_{oe}, q_e)} \leq (1 + \lambda_l)$, where $\lambda_l$ is called the local-optimality threshold.

2. **Global Safety** $\forall q_a \in S$ such that $q_a \neq q_e$; $\frac{c(P_{re}, q_a)}{c(P_{oe}, q_a)} \leq (1 + \lambda_g)$, where $\lambda_g$ is the global stability threshold.

3. AggSERF metric is maximized.

7.2 Robust Plan Generation

Now that we have established the notion of stable plans, we next focus on generating such plans. There have been both intrusive as well as non-intrusive algorithms to generate robust plans. SEER[4] algorithm is a non-intrusive algorithm that generates robust plans without making any changes to the query optimization process, whereas Expand[2] is an algorithm that generates robust plans at compile time itself.

7.2.1 SEER algorithm

SEER[4] is an algorithm that finds out the replacement plans which are robust from among the plans that are present in POSP[9]. This algorithm is non-intrusive and treats the optimizer as the black-box. The costs of plans present in the POSP[9] are evaluated at all the points in the selectivity space $S$ and robust plans are selected from among them. SEER requires a mechanism called *Foreign Plan Costing (FPC)* which finds out the cost of plans in their non endo-optimal regions.
7.2.2 Expand

Expand\cite{2} family of algorithms is based on expanding the candidate set of plans choices retained for the groups in the enumeration lattice, based on both cost and robust criteria. Instead of just having optimal plan for a group, a train of plans is retained, the optimal being called the “train” and the remaining called as “wagons”. The candidate wagons for a group are retained based on following checks:

1. **Local Cost Check** The plans that have local cost within \((1 + \lambda)\) times the cost of optimal plan are retained.

2. **Global Safety Check** In this check the “safety function” is evaluated, which is defined as follows:

\[
f(q_a) = c(p_w, q_a) - (1 + \lambda_g)c(p_e, q_a)
\]

If the function for a plan is less than 0 for all the query points in the selectivity space \(S\), then that plan is retained. A simple heuristic for the same was to evaluate the safety for all the corner points, that is,

\[
\forall q_a \in Corners(S), f(q_a) \leq 0
\]

3. **Global Benefit Check** The above checks ensure that the wagons ensure bounded harm, but does not address the issue of benefit that could be achieved if the engine was to be replaced by a wagon. The benefit of the replacement is addressed by computing the ratio of the arithmetic mean of the costs at the corners of \(S\) of a wagons and the engine.

\[
\xi(p_w, p_e) = \frac{\bar{c}(p_e, q_a)}{\bar{c}(p_w, q_a)}, q_a \in Corners(S)
\]
The wagons having the benefit index greater than 1 are retained as candidate wagons.

4. **Cost-Safety-Benefit Skyline Check** A wagon \( p_{w1} \) is dominated by another wagon \( p_{w2} \) if its local cost is higher, corner costs are individually higher and the global benefit index is lower than \( p_{w2} \). As we move up the enumeration lattice their cost and benefit index come closer together, but the domination property still holds. Hence the dominated plans could be eliminated.

From among the Expand family of algorithms, the Node-Expand algorithm carries out the above checks for each group in the enumeration lattice. At the topmost level in the enumeration lattice the plan having the highest benefit index is chosen as the replacement for the compiler estimated optimal plan. We have implemented and analyzed the performance of the Node-Expand algorithm.

### 7.3 Implementation Details

For algorithms described previously following changes we made to the Pyro optimizer code.

#### 7.3.1 Foreign Plan Costing

Both SEER and Expand algorithm require evaluating the cost of plans in their non endo-optimal regions. Many commercial optimizers today have this feature of *Foreign Plan Costing*. However Pyro did not have this support. A support for the same was provided in Pyro. This required major changes in the file LogicalOp.c and PhysicalOp.c, where while generating the cost of each plan at original locations, costing for plans at other query locations is also computed. A support for both internal *Foreign plan costing* and external in form of an API has been provided.
7.3.2 Expanding the candidate set

Expand requires generating wagon plans for each group that follow the checks described before. While generating the original plan, the wagon plans are also generated. The FPC for wagons is also done at the same time. This required major changes in the files LogicalOp.c and PhysicalOp.c.

7.3.3 Selecting the candidate wagons

Expand has the four checks described before, to obtain the candidate wagons for each group. The checks have been implemented in the Pyro optimizer. This required major changes in the file Volcano.c.
Chapter 8

Experiments for Robust Plan Generation

In this section we first describe the experimental framework and then the experiments performed to analyze the performance of the Expand algorithm implemented in Pyro. We also compare the performance with the SEER algorithm. We also analyze the memory overheads and time overheads for the algorithm. In rest of the section Expand would imply Pyro armed with Expand algorithm.

We have implemented the Expand algorithm in Pyro optimizer to analyze the performance of Expand on a top down optimizer. As mentioned previously Pyro has an option for both constrained and unconstrained optimization. For the experiments related to robust plan generation, we would like compiler to generate optimal plan. Also the experiments are not dependent on the cost based pruning in constrained optimization. Hence we use unconstrained optimization for our experiments.

The experiments are performed on TPCH database. In order to have more alternate plans for replacement, indexes were built on all the attributes of all the relations. The experiments are performed on Sun Ultra 24 workstation with 3 GHz processor, 8 GB of main memory running UBUNTU 10.04. The cost-increase threshold in all our experiments are $\lambda_t, \lambda_g = 20\%$.

**Query Template and Plan Diagram** A plan diagram is a color-coded pictorial
enumeration of the plan choices of the optimizer for a parametrized query template over
the relational selectivity space. It basically captures the POSP[9] geometry. For ex-
ample, consider the parametrized 2D sample query template QT1 shown in Appendix.
Selectivity variations on the SUPPLIER and LINEITEM relations in the query template
QT1 are specified through the \texttt{s\_acctbal :varies} and \texttt{l\_extendedprice :varies} predicates,
respectively. The associated plan diagram for the sample query template is shown
in Figure 8.1(a), produced with the PICASSO optimizer visualization tool[1] on Pyro.
Figure 8.1(b) shows the plan diagram generated for the same query template through
optimizer containing expand algorithm.

![Figure 8.1: Plan Diagrams](a) Plan Diagram from original Pyro(23 plans) (b) Plan Diagram from Pyro armed with Expand(15 plans)

8.1 Plan Diagram Characteristics

In this section we describe the characteristics of the plan diagrams generated for various
query templates. The table 8.1 shows the plan diagram performance for the various query
templates. SEER is able to achieve anorexia. Although the number of plans in the plan
diagram for Expand algorithm are less than those present in original plan diagram, we
were not able to achieve anorexia. The table also shows the analysis of the plans in the
Chapter 8. Experiments for Robust Plan Generation

<table>
<thead>
<tr>
<th>Query Template</th>
<th>Number of plans</th>
<th>SEER Plans</th>
<th>Expand Plans</th>
<th>Non-POSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT 1</td>
<td>23</td>
<td>2</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>QT 2</td>
<td>18</td>
<td>10</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>QT 3</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 8.1: Plan Diagram Performance

<table>
<thead>
<tr>
<th>Query Template</th>
<th>SEER MinSERF</th>
<th>SEER AggSERF</th>
<th>Expand MinSERF</th>
<th>Expand AggSERF</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT 1</td>
<td>-0.27</td>
<td>0.22</td>
<td>-0.08</td>
<td>0.32</td>
</tr>
<tr>
<td>QT 2</td>
<td>-0.67</td>
<td>0.02</td>
<td>-0.58</td>
<td>0.12</td>
</tr>
<tr>
<td>QT 3</td>
<td>-0.15</td>
<td>0.29</td>
<td>-0.15</td>
<td>0.43</td>
</tr>
</tbody>
</table>

Table 8.2: Plan Stability Performance

plan diagram generated for Expand algorithm as to presence of the plans in POSP of the query template. From the table it is clear that there are plans generated by Expand that are from non POSP and hence we might be able to achieve performance better than SEER algorithm. For example for QT1 14 out of the 15 plans generated by Expand are from non POSP region for the query template.

8.2 Plan Stability Characteristics

We have analyzed the plan stability performance in this section for the plan replacement algorithms. Both SEER and Expand algorithm show AggSERF value greater than 0 for all the query templates. This implies that the replacement was beneficial. We are getting AggSERF values for Expand algorithm greater than SEER algorithm for all the query templates. We were able to achieve more robust plans by Expand as compared to those by SEER.

8.3 Computation overheads

Now that we have shown that the Expand algorithm causes improvement in the quality of the plans generated. In this section we show the trade off the quality of the plans
Table 8.3: Computational Overheads

<table>
<thead>
<tr>
<th>Query Template</th>
<th>Time (ms) Pyro</th>
<th>Memory (MB) Pyro</th>
<th>Time (ms) Pyro + Expand</th>
<th>Memory (MB) Pyro + Expand</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT 1</td>
<td>310</td>
<td>6.62</td>
<td>1160</td>
<td>101.21</td>
</tr>
<tr>
<td>QT 2</td>
<td>65</td>
<td>1.59</td>
<td>185</td>
<td>20.39</td>
</tr>
<tr>
<td>QT 3</td>
<td>35</td>
<td>0.91</td>
<td>90</td>
<td>9.21</td>
</tr>
</tbody>
</table>

generated has against the memory and time overhead. The Table 8.3 shows the overhead caused by the Expand algorithm over the normal optimization process. From the table it could be established that the overheads are acceptable.
Chapter 9

Conclusion

In this report we analyzed the traversal of enumeration lattice for various queries. We tried to optimize this process of traversal. We saw evidences for the cases where the traversal could be improved by a factor of at least 2. Each traversal of the enumeration lattice could be associated to a seed. Our aim was to find an optimal traversal of the enumeration lattice. We attacked the problem on two fronts.

In first case we tried to find out an optimal seed among the left-deep seeds. We were able to provide a solution for the problem based on two characteristics of the relations in the query. We provided heuristic based on the degree of relations in the query and the indexes on the relations participating in the query. However we were not able to combine the two heuristics.

In the second case we changed the traversal of enumeration lattice internally by changing the sequence of logical expressions in groups. We obtained good results for clique and star queries. In some cases we were able to provide performance better than the optimal left-deep seed. For cyclic and chain queries we were able to provide robust traversal by reducing the range of values of total visits to the nodes in enumeration lattice.

A further line of work could be that we need to find a good heuristic for all types of queries.

We have also implemented the Expand algorithm in the top down optimizer Pyro,
which generates robust plans for a query that are relatively less sensitive to the selectivity errors. We have also analyzed the stability achieved through the replacement. We were able to achieve plans that are more resistant to selectivity errors.
Appendix A

Appendix

A.1 Query Templates

1. QT 1 (TPCH 8)
   select n2.name,o.orderdate,l.extendedprice
   from part,supplier,lineitem,orders,
   customer,nation1,nation2,region
   where p.partkey=l.partkey
   and s.suppkey=l.suppkey
   and l.orderkey=o.orderkey
   and o.custkey=c.custkey
   and c.nationkey = n1.nationkey
   and n1.regionkey=r.regionkey
   and r.name=4
   and s.nationkey=n2.nationkey
   and o.orderdate < 10720
   and p.type = 75
   and s.acctbal :varies
   and l.extendedprice :varies
   order by n2.name
2. QT 2 (TPCD1)

```sql
select max(o_custkey)
from customer, orders, lineitem, supplier, nation, region, partsupp
where o_orderkey = l_orderkey
and l_suppkey = s_suppkey
and c_name = s_name
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and s_acctbal = ps_supplycost
and l_extendedprice :varies
and c_acctbal :varies
group by n_nationkey;
```

3. QT 3 (TPCD2)

```sql
select max(c_custkey)
from customer, orders, lineitem, supplier, nation, part
where o_orderkey = l_orderkey
and o_custkey = c_custkey
and o_orderdate > 18400
and c_nationkey = n_nationkey
and l_partkey = p_partkey
and s_acctbal = p_retailprice
and l_extendedprice :varies
and c_acctbal :varies
group by n_nationkey;
```
A.2 Type of query graphs

We have used four types of query graphs in our report. We will give an example of each in this section. Given a query, if we consider the relations as the vertices and joins in the where clause as the edges, the graph that is formed from the query classifies the query. The four types of query graphs are as follows:

1. Chain Query: The join graph is linear and forms a chain
   
   ```
   select l.extendedprice 
   from nation1, nation2, supplier, orders, 
   lineitem, customer, region 
   where n2.nationkey = s.nationkey 
   and s.suppkey = l.suppkey 
   and l.orderkey = o.orderkey 
   and o.custkey = c.custkey 
   and c.nationkey = n1.nationkey 
   and n1.regionkey = r.regionkey
   ```

2. Cyclic query: The join graph forms a cycle
   
   ```
   select l.extendedprice 
   from nation1, nation2, supplier, orders, 
   lineitem, customer 
   where n2.nationkey = s.nationkey 
   and s.suppkey = l.suppkey 
   and l.orderkey = o.orderkey 
   and o.custkey = c.custkey 
   and c.nationkey = n1.nationkey 
   and n1.regionkey = n2.regionkey
   ```

3. Star query: The join graph forms a star
   
   (a) select p.partkey
from nation1, supplier, lineitem, orders, customer, part
where s_name = p_selkey
and s_suppkey = o_comment
and s_nationkey = n1_nationkey
and s_selkey = c_name
and s_comment = l_suppkey

(b) select p_partkey
from supplier, lineitem, orders, customer, part, partsupp
where l_orderkey = o_orderkey
and l_partkey = p_partkey
and l_quantity = ps_availqty
and l_suppkey = s_suppkey
and l_extendedprice = c_acctbal

(c) select p_partkey
from nation1, supplier, lineitem, orders, customer, part, partsupp
where c_custkey = o_custkey
and c_name = p_name
and c_address = s_address
and c_nationkey = n1_nationkey
and c_acctbal = l_extendedprice
and c_comment = ps_comment

4. Clique query: The join graph is completely connected
select p_partkey
from part, supplier, customer, partsupp, lineitem
where p_partkey = ps_suppkey
and p_name = s_comment
and p_selkey = l_partkey
and p_retailprice = c_acctbal
and ps_partkey = s_suppkey
and ps_availqty = l_quantity
and ps_selkey = c_name
and s_selkey = l_suppkey
and s_name = c_comment
and l_orderkey = c_custkey
References


