Analyzing the Behavior of a Distributed Database Query Optimizer

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

by

Deepali Nemade

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

June 2013

©Deepali Nemade

June 2013

All rights reserved

TO

My Parents

for their support and encouragement

Acknowledgements

At times our own light goes out and is rekindled by a spark from another person.
Each of us has a cause to think with deep gratitude of those who have lightened the flame within us.
-"Albert Schweitzer"

With all my heart I would like to express my great appreciation to Prof. Jayant R. Haritsa for providing his invaluable guidance throughout this work. I have learnt a lot during this journey and he was also there to guide, to help and to teach.

Many thanks to all my labmates for their help and suggestions. I would like to offer my special thanks to Anirudh Kushwah for his suggestions and encouragement. I thank my family for their support and encouragement during my stay at IISc. And last but not the least I thank all my friends for making the journey in IISc special and memorable.

- Deepali Nemade

Abstract

Performance of a database system depends on the efficiency of its optimizer. Query optimizers of distributed databases are designed to process complex queries over distributed data and need to explore opportunities of parallelism in query execution while choosing an optimal plan for a given query. This makes overall design of these optimizers complex as compared to centralized optimizers. But with complex functionalities, high performance is also desirable. Therefore it is important to analyze their behavior. In this work, we have done analysis of a distributed database query optimizer COM_OPT using the tool Picasso [1], which using metadata statistics stored in database catalogs shows a suite of diagrams that captures the behavior of optimizer. With our analysis over these diagrams we have shown that the behavior of COM_OPT deviates from the expected behavior, i.e. it often makes fine grained plan choices across the selectivity space, does not follows the assumptions of PQO [2] and also shows non monotonically increasing cost behavior with increasing selectivities of the base relations.

There are various other testing applications also which works solely with metadata statistics. But effective testing needs alternative metadata scenarios. A tool "The CODD Metadata Processor" provides an interface for ab-initio construction of such futuristic metadata scenarios. Since distributed databases are designed to handle huge data, testing of their optimizers should also be done using alternative metadata scenarios corresponding to big data. But generating such metadata scenarios by constructing corresponding data scenarios and then collecting statistics is infeasible. Therefore, to make the construction of these futuristic metadata scenarios feasible and simpler, we have engineered the tool CODD [5], which provides an interface for constructing alternative metadata

scenarios, to support COM_OPT. Since COM_OPT supports construction of statistics over group of columns along with individual columns, we have also included multi-column histogram support in the tool. Also, we have redesigned CODD with 3-tier client-server architecture.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**
1.1 Organization 3

2 Visualizing the behavior of query optimizers **4**
2.1 Optimizer Diagrams 6

3 Introduction to COM_OPT: A distributed database query optimizer **12**
3.1 Schematic Information Storage in COM_OPT 12
3.2 Histogram Statistics 13
3.3 Extracting Query Execution Plans 14
3.4 INFORMATION EXTRACTION FROM COM_OPT 14
3.5 Accessing metadata statistics 15
3.6 Extracting Plan information 15

4 Result Analysis **16**
4.1 Implementation Details 16
4.2 Testbed Environment 16
4.3 Analysis of diagrams for Query Template 21 18
4.3.1 Plan Diagram 18
4.3.2 Compilation Cost Diagram 19
4.3.3 Reduced Plan Diagram 19
4.3.4 Plan Tree 20
4.4 Interesting Picasso Art Gallery 22
4.5 Comparison with centralized db optimizers 25

5 Making construction of futuristic metadata scenarios easier **27**

6 The CODD Metadata Processor **29**
6.1 Our Contributions 32

7 Redesigning CODD Architecture **33**
7.1 Motivation . . . 33
7.2 Architectural Overview . . . 34

8 Engineering CODD for COM_OPT **37**
8.1 Metadata Construction . . . 37
8.1.1 Ab-Initio metadata construction . . . 38
8.1.2 Metadata Validation . . . 39
8.1.3 Implementation Details . . . 41
8.2 Metadata Retention . . . 43
8.2.1 Implementation Details . . . 44
8.3 Metadata Scaling . . . 44
8.3.1 Size-based Scaling . . . 45
8.3.2 Time-based Scaling . . . 45

9 Conclusions **47**

A **48**

References **58**

List of Tables

8.1 Consistency Constraints 42
8.2 Cost of queries before and after scaling 46

List of Figures

1.1 Overview 2

2.1 Query Template 2 of TPC-H Benchmark 5
2.2 Plan Diagram 7
2.3 Compilation Cost Diagram 8
2.4 Compilation Cardinality Diagram 8
2.5 Reduced Plan Diagram 9
2.6 Plan Tree 10
2.7 Plan Tree Difference (Plan P1 and Plan P4) 11

4.1 Query Template 21 of TPC-H Benchmark 17
4.2 Plan Diagram 18
4.3 Compilation Cost Diagram 19
4.4 Reduced Plan Diagram 20
4.5 Plan Tree for Plan P2 21
4.6 Plan Diagram for QT 8 22
4.7 Plans differing in the position of EXP_EXCHANGE operator 23
4.8 Difference in group by operation 24
4.9 Comparison with centralized optimizers 25

5.1 Conventional Method for Generating Metadata 27
5.2 Time and Space Overheads in Conventional Process 28
5.3 CODD for Constructing Metadata Statistics 28

6.1 Construct Mode 30
6.2 Retain Mode 30
6.3 Scale Mode 31

7.1 Earlier 2-tier CODD Architecture 33
7.2 CODD 3-tier architecture 35
7.3 Working of CODD 3-tier Architecture 36

8.1 Construct Mode interface for COM_OPT 38
8.2 Constraint Graph 40
8.3 Frequency Value Histogram 41

8.4 Construct Mode Implementation Overview 43
8.5 Time Scaling Overview 45

Chapter 1

Introduction

Query optimizer is an integral part of relational database system that tries to identify best execution plan for efficient execution of declarative SQL queries. The performance of a database system depends on the effectiveness of its query optimizer and is measured in terms of "cost" which indicates the total execution time of the queries. An efficient optimizer should ideally come up with the minimum cost execution plan. The role of query optimizers has become especially critical due to high degree of query complexity characterizing current data warehousing and mining applications, as exemplified by the TPC-H decision support benchmark [10]. Thus, analysis of optimizer's behavior is extremely important. Specifically, distributed database systems are designed to process complex queries over large amount of data which is distributed across various nodes. Also, data may be replicated across the nodes. Optimizers in these systems face the highly complex problem of determining the best execution plan in the presence of multiple possible parts of relation copies located at different nodes. The execution site for each operator has to be determined and the cost of transferring relations between sites has to be included in the cost model of the optimizer in addition to the I/O's performed. The optimizer must be able to recognize the opportunities of parallelism that arise as a result of distributed settings and adjust its cost model to recognize plans that utilize these possible concurrent execution opportunities. Despite complex functionalities, the optimizer must provide efficient execution plans. Thus, analysis of distributed database

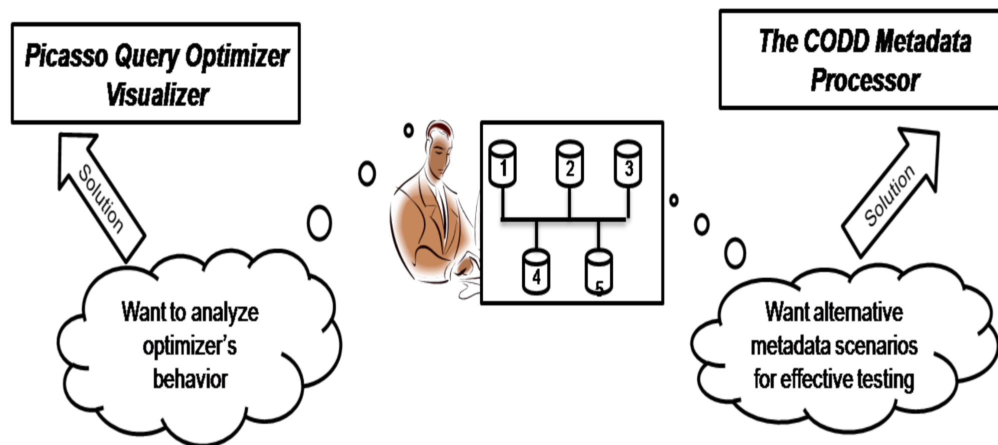

Figure 1.1: Overview

query optimizers is extremely important. The tool "Picasso Query Optimizer Visualizer" introduced in [8] helps in visualizing the behavior of optimizers. With the help of this tool, analysis of query optimizers of leading commercial centralized database systems has been done in past. It was shown that their optimizers often violates the expected behavior. This has motivated us to analyze the behavior of a distributed database query optimizer. So, as a part of this work, we have done the analysis of query optimizer of a distributed database system. We will call this optimizer as COM_OPT.

Picasso analyzes the behavior of optimizer using metadata statistics only. There exists other testing applications also for which inputs are solely based on metadata statistics stored in the database. But creating one such metadata scenario requires creation of corresponding data, loading it into the database and then collecting statistics over it. But time and space overheads involved in this process makes it infeasible. Distributed databases are designed to handle huge amount of data. So, their testing should also be done on metadata scenarios corresponding to big data. For effective testing of their optimizers, one needs to create alternative metadata scenarios corresponding to big data. But creating various such big data scenarios is infeasible and thus testing process becomes restricted. Recently developed tool "The CODD Metadata Processor" [6] attempts to alleviate these difficulties through the construction of "Dataless Databases". Using CODD,

metadata can be easily constructed within a matter of minutes, including the desired attribute-value distribution through visual histogram construction. Thus for constructing alternative futuristic metadata scenarios for a distributed database COM_OPT, we have engineered the tool CODD to support COM_OPT. In addition to this, we have also done an architectural enhancement to the tool by redesigning it with 3-tier client-server architecture. The tool was initially implemented as a 2-tier application.

1.1 Organization

Rest of the report is organized as follows: Chapter 2 gives a brief overview of the optimizer diagrams and their utilization. Chapter 3 provides introduction to the distributed database COM_OPT and also desceibes how to extract required metadata statistics from the database. Chapter 4 provides implementation details and visual analysis of COM_OPT using optimizer diagrams. Chapter 6 gives a brief introduction to the tool CODD. Chapter 7 describes the 3-tier architecture of the tool. Chapter 8 contains the details of engineering CODD for supporting COM_OPT.

Chapter 2

Visualizing the behavior of query optimizers

For a given database and a system configuration, the query optimizer's execution plan choices are primarily a function of the *selectivities* of the base relations in the query. The selectivity of a relation is the estimated fraction of rows of the relation that are relevant for producing the final result. In [4], the concept of *optimizer diagrams* was introduced to denote the color-coded pictorial enumerations of the plan choices of the optimizer for parameterized SQL queries over the relational selectivity space. For example, consider Query Template 2 shown in Figure 2.1, a two-dimensional query template based on Query 2 of TPC-H benchmark, with selectivity variations on PART and PARTSUPP relations through *p_retailprice* $<= c1$ and *ps_supplycost* $<= c2$ range predicates, respectively. Given a query template, Picasso captures the behavior of the optimizer over the selectivity space in a suite of diagrams, which are collectively known as "optimizer diagrams". Picasso uses metadata statistics stored in database catalogs to produce these diagrams. These diagrams helps in visually analyzing the behavior of the optimizer. The visual analysis of the behavior of optimizer across the selectivity space provides various insights, mainly including the following:

- **Granularity of Plan choices:** It is expected that the plans chosen by the optimizer will not change frequently with small changes in the selectivities of relations

```
select
  s_acctbal, s_name, n_name, p_partkey, p_mfgr,
  s_address,s_phone, s_comment
from
  part, supplier, partsupp, nation, region
where
  p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_retailprice <= c1
  and s_nationkey = n_nationkey
  and n_regionkey = r_reigonkey
  and r_name = 'EUROPE'
  and ps_supplycost <= (
    select
      min(ps_suppcost)
    from
      partsupp, supplier, nation, region
    where
      p_partkey = ps_partkey
      and s_suppkey = ps_suppkey
      and s_nationkey = n_nationkey
      and r_name = 'EUROPE'
      and ps_supplycost <= c2
  )
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey
```

Figure 2.1: Query Template 2 of TPC-H Benchmark

involved in the query.

- **Validity of Parametric Query Optimization (PQO)**: The concept of PQO [2] tries to identify a set of optimal plans at compile time, which can subsequently be used at run time to identify best plan when the actual parameter values are known. The expectation is that this would be much faster than optimizing query from scratch. But the assumption is that the plan regions should be unique, homogeneous and convex and from implementation perspective the number of plans across the selectivity space obtained at compile time must be less. In the diagrams we will see if the optimizer follows the assumptions of PQO or not.

- **Plan Cost Monotonicity (PCM)**: With a cost based query optimizer having linear cost model, it is expected that costs will increase along with the increasing selectivities. With the help of diagrams we will analyze the cost behavior of the optimizer.

Analysis of various leading centralized database query optimizers was done in [4]. It was observed that the behavior of optimizer often deviates from the expected behavior. The optimizers often makes fine grained plan choices across the selectivity space. It was also shown that the assumptions of PQO do not hold in practice. Various instances were shown where PCM violation takes place across the plans and surprisingly within the plans as well. While analyzing the behavior of COM_OPT we will see if it shows the expected behavior or not by observing above mentioned features. Also, we will see if it is showing some anomalous behavior.

Following section provides a brief introduction to the optimizer diagrams produced by Picasso. Detailed implications of the diagrams will be discussed in Chapter 4.

2.1 Optimizer Diagrams

For capturing the plan choices, cost and cardinality estimation, etc. of the optimizer, Picasso produces a suite of optimizer diagrams. Figures in this section show a representative set of optimizer diagrams produced by Picasso (for COM_OPT) for Query Template 2 with exponentially distributed query points (300 Resolution).

Plan Diagram: Plan diagram denotes a color-coded pictorial enumeration of the execution plan choices of a database query optimizer for a query template over the selectivity space. The plan diagram for QT 2 is shown in Figure 2.2, where X and Y axis determine the percentage selectivities of PART and PARTSUPP relations respectively, and each color-coded region represents a particular plan that has been determined by the optimizer to be the optimal choice in that region. Plan diagram shows that a set of 137 plans, P1

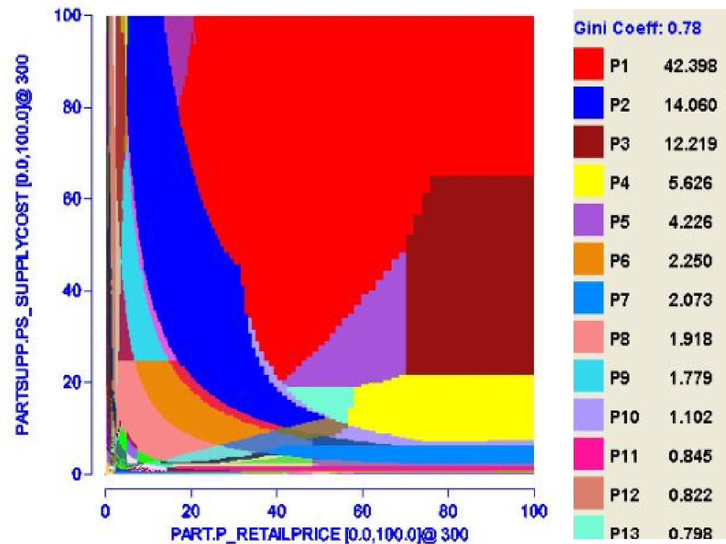

Figure 2.2: Plan Diagram

through P137, covers the entire selectivity space. The value associated with each plan in the legend indicates the percentage area coverage of that plan in the diagram. And the Gini coefficient represents the skew in the plan areas. In this plan diagram the Gini Coefficient [7] of 0.78 shows that there are very less number of plans which are covering a major portion of the selectivity space, while most of the plans are having very less area.

Compilation Cost Diagram: Complementary to the plan diagram is the cost diagram. Cost diagram for Query Template 2 is shown in Figure 2.3. It gives a visualization of the estimated plan execution costs over the relational selectivity space. X axis and Y axis represent the selectivity variations of the relations while the Z axis represents cost (normalized with respect to the maximum cost over the space). This diagram helps in understanding the variation of cost within a plan and across plans for a query template over the selectivity space. The cost diagram here shows non monotonically increasing cost behavior (PCM violation) with increasing selectivities near low selectivity region, which is not expected from a cost based optimizer.

Compilation Cardinality Diagram: It is similar to compilation cost diagram except

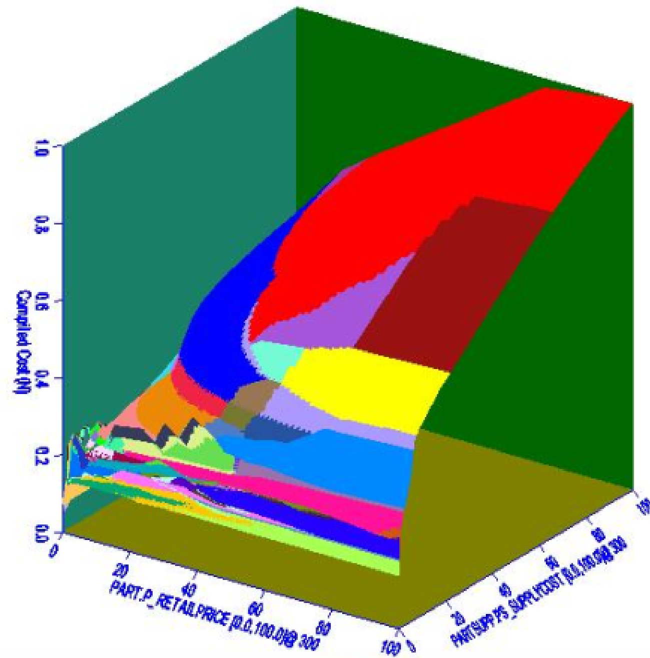

Figure 2.3: Compilation Cost Diagram

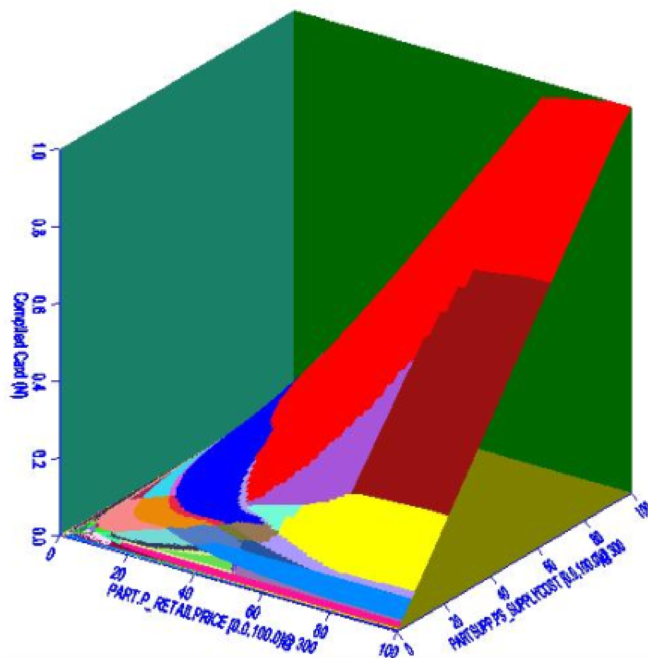

Figure 2.4: Compilation Cardinality Diagram

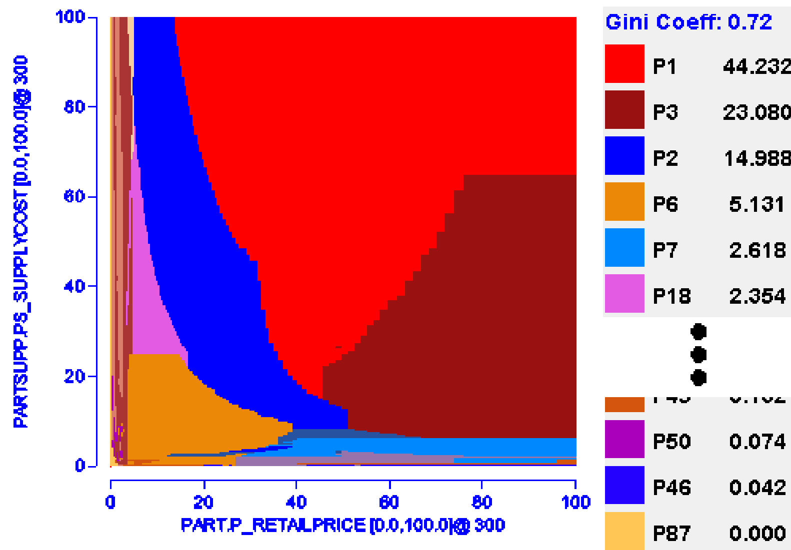

Figure 2.5: Reduced Plan Diagram

that it shows the variation of optimizer's estimation for cardinality of result set over the selectivity space instead of cost of execution. The cardinality diagram for QT 2 is shown in Figure 2.4. The estimated cardinality result here increases along with increasing selectivities as expected.

Reduced Plan Diagram: It shows the extent to which the original plan diagram may be simplified (by replacing some of the plans with their siblings in the plan diagram) without increasing the cost of any individual query by more than a user specified threshold. One such strategy is "cost greedy reduction" which we have used here. In this reduction, for any point x, only the plans which lies in its first quadrant are considered as candidates for swallowing the plan at that point, since it is expected that these points will have cost greater than that of query point x as they have to process more number of tuples. Figure 2.5 shows the reduced plan diagram for QT2 at 20% cost increase threshold. The number of plans are now 15, which is much less than that in original plan diagram (137 plans). The reduced plan diagram is near to anorexic plan diagram (having around 10 plans) and thus for this diagram the concept of PQO is feasible from implementation point of view.

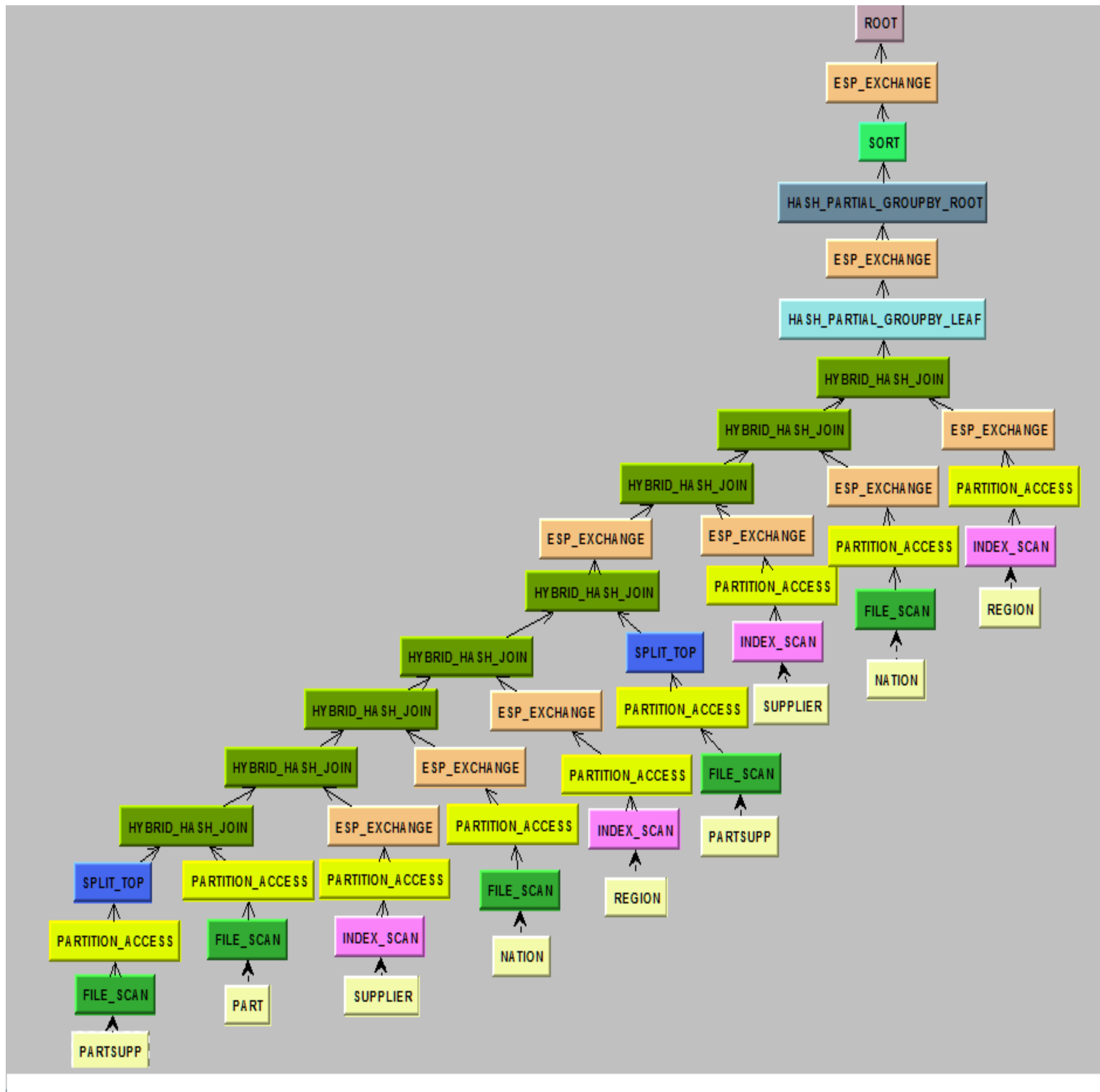

Figure 2.6: Plan Tree

Plan Tree: It is a tree representation of execution plan chosen by the optimizer for given selectivities of the relations. A sample plan tree for Query Template 2 is shown in Figure 2.6. Picasso also shows Compiled Plan Tree which additionally shows the cost and/or cardinality associated with each operator in the plan tree.

Plan Tree Difference: It highlights the schematic differences between a selected pair

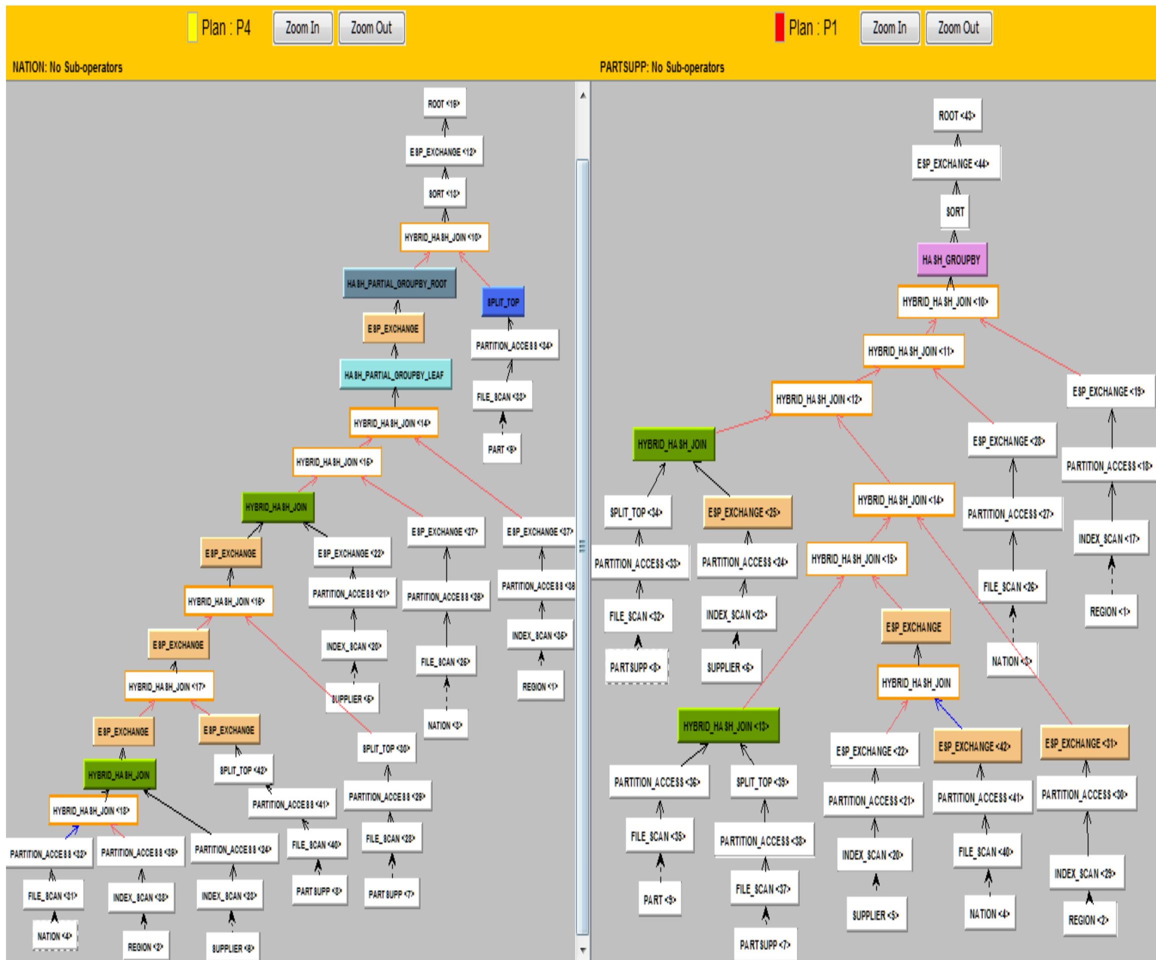

Figure 2.7: Plan Tree Difference (Plan P1 and Plan P4)

of plans in the plan diagram. Plan tree difference between plan P1 and P4 is shown in Figure 2.7.

Chapter 3

Introduction to COM_OPT: A distributed database query optimizer

In a distributed database, data is distributed across various nodes. To keep track of the data, it stores schema and metadata information about data of relations spread across various nodes. This chapter describes how the schematic information is stored in COM_OPT, how to extract distribution statistics corresponding to an attribute and the way to extract a query execution plan from COM_OPT.

3.1 Schematic Information Storage in COM_OPT

The database system maintains various catalogs which act as containers for schema's. A schema contains a collection of database objects like *tables, views* and *indexes.* The user defined tables are stored in their respective user catalogs and schemas. Along with user defined tables, within the same schema histogram tables are also stored which contains metadata statistics for the relations stored in that schema. The database also maintains a system catalog which contains six schemas within them. One such schema is DEFINITION_SCHEMA_VERSION_

vernum, which stores additional metadata (e.g. schema information, constraints, etc.) associated with each object in the catalog. SYSTEM_SCHEMA table resolves object names given object identifier. SYSTEM_DEFAULTS_SCHEMA table stores system default settings. Users can change the default settings for externalized attributes in the SYSTEM_DEFAULTS table in the system catalog. OBJECTS and COLS tables in this schema stores the structural details of relations and their columns respectively. The datatype of each column of a relation is stored in COLS table. A thing to note here is that the system metadata tables are not user updatable.
The optimizer works at Optimization levels 0(minimum), 2, 3(default) or 5(maximum). The default value of the OPTIMIZATION_LEVEL can be changed for the current session using the CONTROL QUERY DEFAULT statement.

3.2 Histogram Statistics

The optimizer uses histograms to estimate the number of tuples that come out of each operator in a query execution plan and uses these estimates to calculate the total cost of a plan. In COM_OPT, histogram statistics for the entire table and columns are stored in the HISTOGRAMS and HISTOGRAM_INTERVALS user metadata tables, which are stored in user catalog. HISTOGRAMS table stores relation level (e.g. relation cardinality) and column level (e.g. number of unique values, High Value, Low Value, etc.) metadata information's while HISTOGRAM_INTERVALS table stores the interval information for the data distribution of a column or a group of columns. An interval represents a range of values for the column. The range of values for each interval is selected such that every interval represents approximately the same number of rows in the table. This is known as "equi height" distribution of the values over the histogram intervals.
It is important to note that the histogram tables are not automatically updated when you update a relation for which statistics are stored or when a new relation is created. To keep the histogram statistics current, UPDATE STATISTICS statement has to be

executed after creating or updating the relations.

3.3 Extracting Query Execution Plans

COM_OPT generates alternative plans and chooses the best execution plan for the query by using its cost model. The optimizer computes the cost of each alternative and chooses the one with the lowest cost as the optimal query execution plan. EXPLAIN function can be used for extracting the execution plan information for a given query.

EXPLAIN function: The EXPLAIN function is a table-values stored function that returns information about execution plans for SQL DML statements. One can query and display certain columns or all columns of information about execution plan. Two steps are required to get the execution plan for a query:

1. Prepare Statement:
   `Prepare` *label* `from` *query*
2. EXPLAIN function:
   `select * from EXPLAIN(`*label*`, NULL);`

3.4 INFORMATION EXTRACTION FROM COM_OPT

For producing the diagrams which characterizes optimzier's behavior, Picasso needs following things from the optimizer:

1. Extraction of distribution statistics for varying predicates of the query template for finding out the constants for them at each selectivity point to form queries with it
2. Extraction of relevant portions of execution plans for each query in the selectivity space.

3.5 Accessing metadata statistics

The statistics used for producing various diagrams are cardinality of relation, attributes of the table with their types and distribution of data in a column. Metadata statistics information is stored in HISTOGRAMS and HISTOGRAM_INTERVALS tables which can be extracted using the schematic information stored in system metadata tables. HISTOGRAMS table stores the relation level and column level statistics, while HISTOGRAM_INTERVALS table stores distribution of values in a column of group of columns. These information are extracted as follows:

1. Corresponding to each TABLE_UID which is a unique identifier for a relation, HISTOGRAMS table stores ROWCOUNT which is the cardinality of the relation. The DEFINITION_SCHEMA_VERSION_*vernum*.OBJECTS table stores the mapping of the relation names to their unique id's.

2. The datatype of an attribute can be extracted from the table DEFINITION_SCHEMA _VERSION_*vernum*.COLS using column name and the unique identifier for the corresponding relation to which the attribute belongs.

3. The distribution statistics for an attribute or group of attributes can be extracted from HISTOGRAM_INTERVALS table.

3.6 Extracting Plan information

EXPLAIN function can be used for extracting the relevant information about the query execution plan for a given query.

Chapter 4

Result Analysis

4.1 Implementation Details

Picasso is written in JAVA. It follows 3-tier client server architecture where the client provides a vendor neutral interface to the users while for each database engine there is a separate module for extracting information's discussed in above section. So, for engineering Picasso for COM_OPT, along with some additions in few portions of the code on client side, we have added a new engine specific module on the server side. The step by step procedure of porting, given in the Picasso user manual, was followed for this purpose. We have written approximately 2K lines of code for the purpose of engineering Picasso for supporting COM_OPT.

4.2 Testbed Environment

1. Database: TPC-H (1 GB scale) representing a manufacturing environment with statistics collected on all columns of all relations. Indexes were created on attributes of the relations.

2. Query Set: Query Templates based on TPC-H benchmark queries.

3. Computational Platform: Non Stop operating System, Intel Itanium Quad Core

```
select
  s_name, count(*) as numwait
from
  supplier, lineitem l1, orders, nation
where
  s_suppkey = l1.l_suppkey
  and o_orderkey = l1.l_orderkey
  and o_orderstatus = 'F'
  and exists (
   select * from lineitem l2
   where
     l2.l_orderkey = l1.l_orderkey
     and l2.l_suppkey <> l1.l_suppkey
  ) and not exists (
   select * from lineitem l3
   where
     l3.l_orderkey = l1.l_orderkey
     and l3.l_suppkey <> l1.l_suppkey
     and l3.l_receiptdate > l3.l_commitdate
 )
  and s_nationkey = n_nationkey
  and s_acctbal <= c1
  and l1.l_extendedprice <= c2
  and n_name = 'SAUDI ARABIA'
group by
  s_name
order by
  numwait desc, s_name
```

Figure 4.1: Query Template 21 of TPC-H Benchmark

processor 1.6 GHz, 16GB RAM.

4. Picasso Settings: Diagram resolution was set to 300 and the distribution of query points was set to exponential.

Most importantly the data was physically distributed across various nodes. The partitioned DDL's of the relations can be found in Appendix.

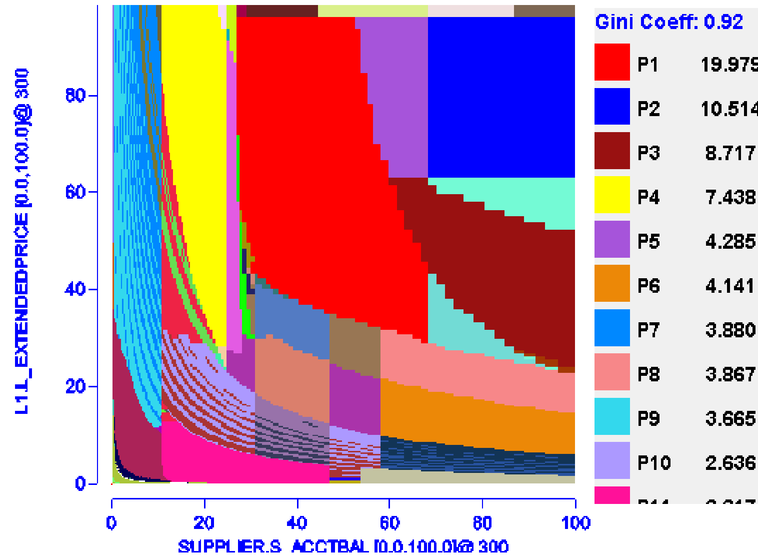

Figure 4.2: Plan Diagram

4.3 Analysis of diagrams for Query Template 21

Figure 4.1 shows Query Template 21 based on query 21 of TPC-H Benchmark with selectivity variation on SUPPLIER and LINEITEM relations through $s_acctbal <= c1$ and $l_extendedprice <= c2$ range predicates respectively. In this section we will focus our analysis over the various diagrams produced by Picasso for this query template.

4.3.1 Plan Diagram

Figure 4.2 shows the plan diagram for Query Template 21. A set of 131 different optimal plans, P1 through P131, covers the entire selectivity space. The legend shows that plan P1 is covering 19% of the space, whereas plan P5 onwards have less than 5% area each. Also from the diagram it can be seen that the optimizer has made extremely fine grain choices near the low selectivity region. Towards high selectivity regions also the plans are not covering very large areas. There are plan areas which are not homogeneous and convex. For example, plan P7 and P9 are occurring alternatively in region near the vertical axis. The plans here violates the assumption of PQO that plan regions must be convex.

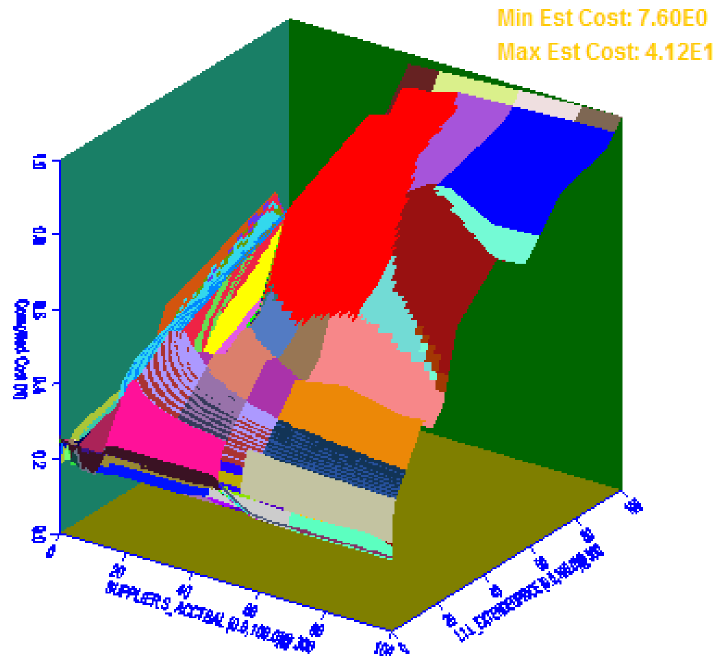

Figure 4.3: Compilation Cost Diagram

4.3.2 Compilation Cost Diagram

The compilation cost diagram for Query Template 21 is shown in Figure 4.3. The cost diagram shows that the overall cost of query execution increases along with increasing selectivities across the selectivity space. But there are regions where the plan cost is not monotonically increasing with increasing selectivities. This behavior can be seen within the plan and across the plans as well. Such behavior is not expected from a cost based optimizer.

4.3.3 Reduced Plan Diagram

The reduced plan diagram for above plan diagram is shown in Figure 4.4. The reduction was done using cost greedy reduction algorithm by setting the cost increase threshold λ, to 20% . Here we got a significant reduction but survival of 27 plans shows the lack of anorexic reduction. This is due to violations in the Plan cost monotonicity principle in the plan diagram. It can be seen that many small sized plans are still retained near

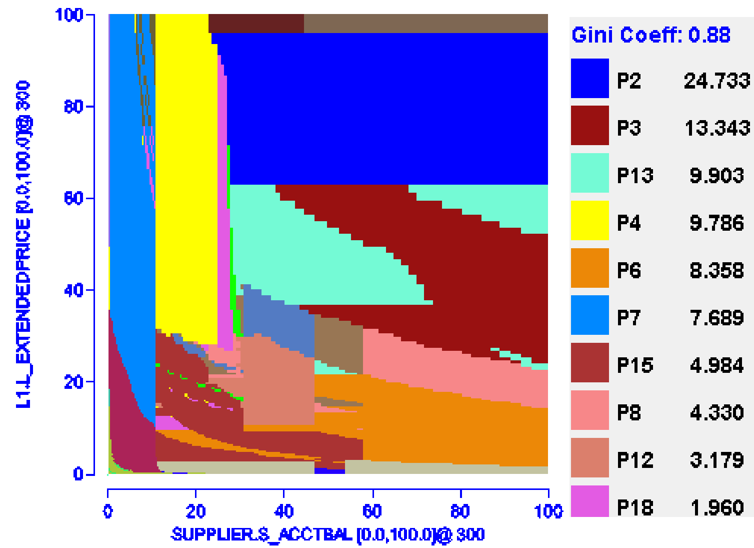

Figure 4.4: Reduced Plan Diagram

low selectivity region, as there were cost monotonicity violations in this region. Here we could have got an anorexic reduced plan diagram, if the reduction was done using the concept of foreign plan costing, i.e. by calculating the cost of each plan (which occurs in first quadrant of the point) at the query point and allowing the replacement of plan only if the new plan is able to swallow full region of the plan corresponding to the query point.

4.3.4 Plan Tree

COM_OPT has a richer set of operators as compared to centralized database optimizers. It has some special operators which are indicative of parallelism in the execution plan. Some of these special operators can be seen in the plan tree for plan P2 shown in Figure 4.5. While executing, the operators will be evaluated in bottom up fashion. Here we will give an overview of the special operators which represents parallelism in the execution plan. Such operators are not available in centralized database query optimizers and are of interest of our study.

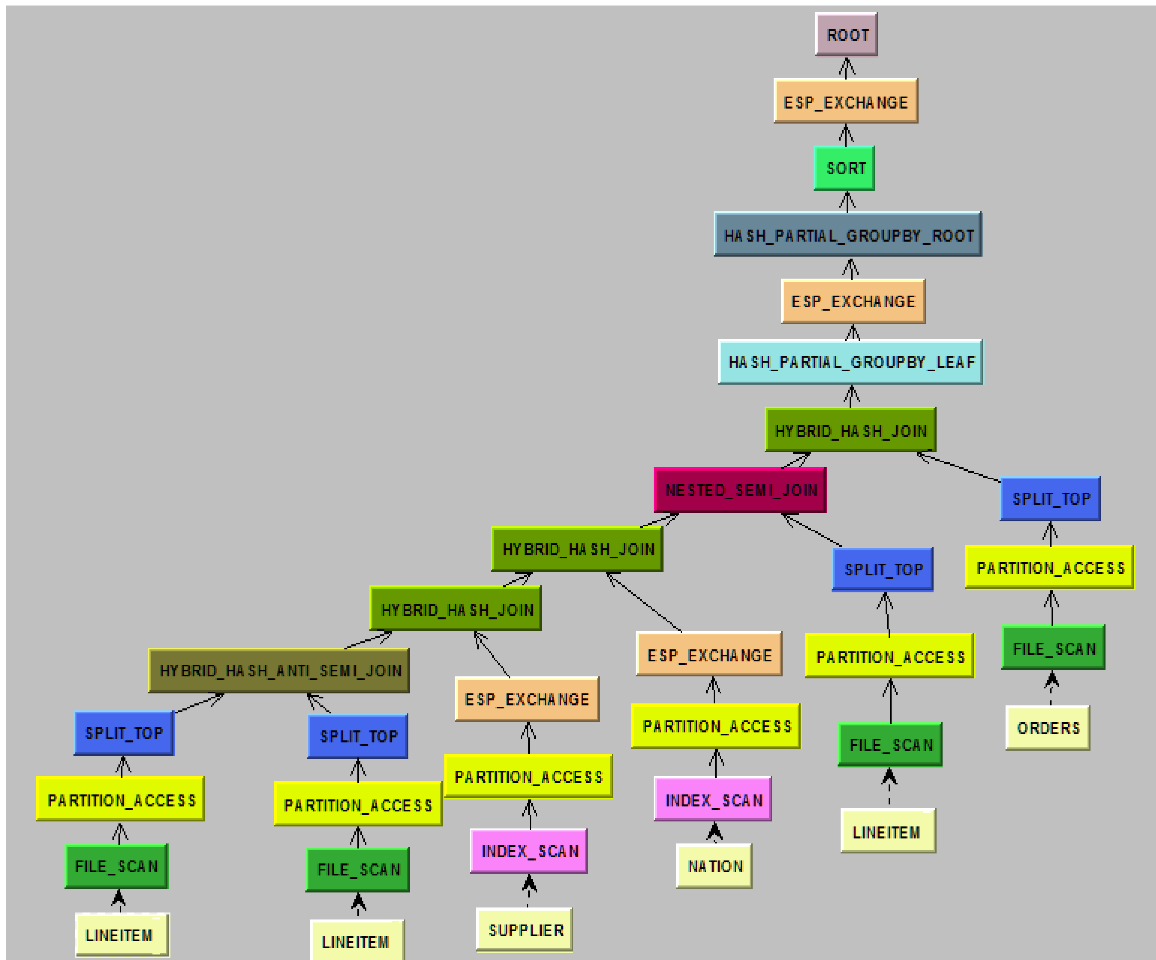

Figure 4.5: Plan Tree for Plan P2

1. PARTITION_ACCESS: It is a unary node that appears in parallel plans but does not signify parallelism. All disk access within a PARTITION_ACCESS node are serial, i.e. it provides access to single data partition of a relation at a time.

2. SPLIT_TOP: This operator occurs above PARTITION_ACCESS operator in plan tree and signifies parallelism in execution plan. It is an N-ary operator that provides access to more than one PARTITION_ACCESS nodes and as a result it facilitates Data Access Manager parallelism.

3. ESP_EXCHANGE: This operator can occur anywhere in the plan tree. It describes the portion of plan that redistributes the input data stream. It generates multiple ESP's (Executor Server Processes) which performs parallel operation over the input

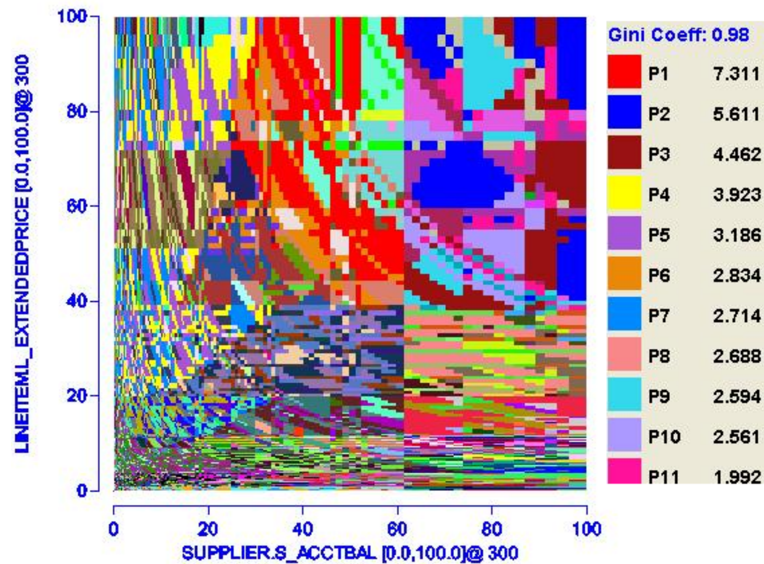

Figure 4.6: Plan Diagram for QT 8

data stream.

4. Partial group by operators: For reducing the amount of data that must be relocated for a query, HASH_PARTIAL_GROUPBY_LEAF operator executes a partial group by operation close to where the data is read. When executed in DAM (Data Access Manager), the group by operation is limited to a small amount of memory. Any group that does not fit in memory is passed on ungrouped, with the full grouping occurring at the HASH_PARTIAL_GROUPBY_ROOT.

4.4 Interesting Picasso Art Gallery

While analyzing the optimizer diagrams for COM_OPT we came across a plan diagram, shown in Figure 4.6, which has gained our special attention because of its extreme density. This diagram was produced for query template 8 which corresponds to query 8 of TPC-H benchmark which is a fairly complex query with dynamic base relations in form of a sub-query and consists of group by and order by operations as well. The sub-query contains two varying predicates and involves joins of 8 relations. The plan

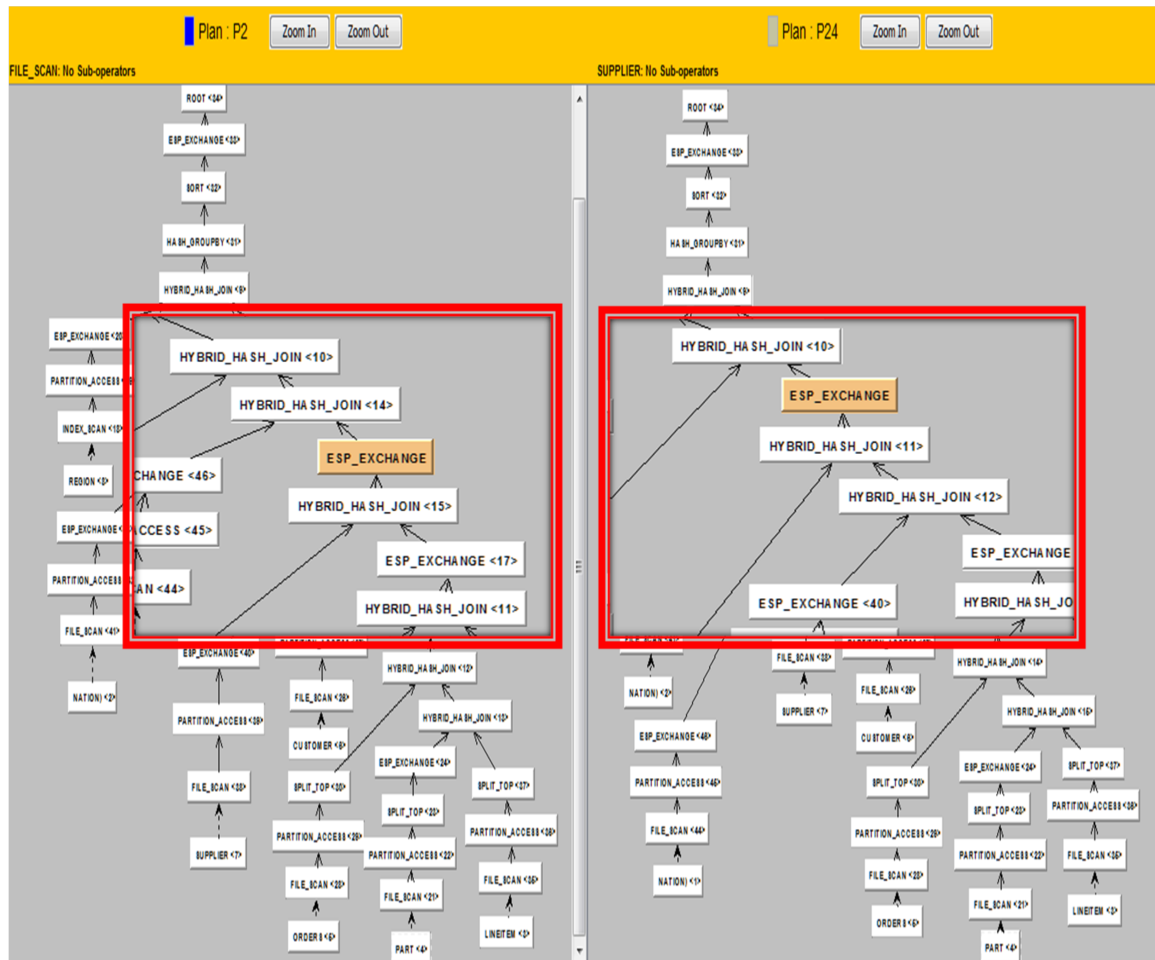

Figure 4.7: Plans differing in the position of EXP_EXCHANGE operator

diagram consists of 2041 plans. Such a highly dense plan diagram was never seen before during the analysis for centralized optimizers. Here the optimizer seems to be making extremely fine grained choices throughout the selectivity space. The plan boundaries are highly irregular. There are several plans (e.g. plan P1, P2) which are occurring in duplicate locations. On closer analysis it was observed that the neighboring plans were not having very significant cost difference.

On further analyzing this plan diagram we found that the total number of different join orders among these plans were 681, which itself is a very high number (much greater than the plan diagram cardinality of any other optimizer). Apart from the join orders, the differences in the plans were also found mainly because of parallelism operators. The plan

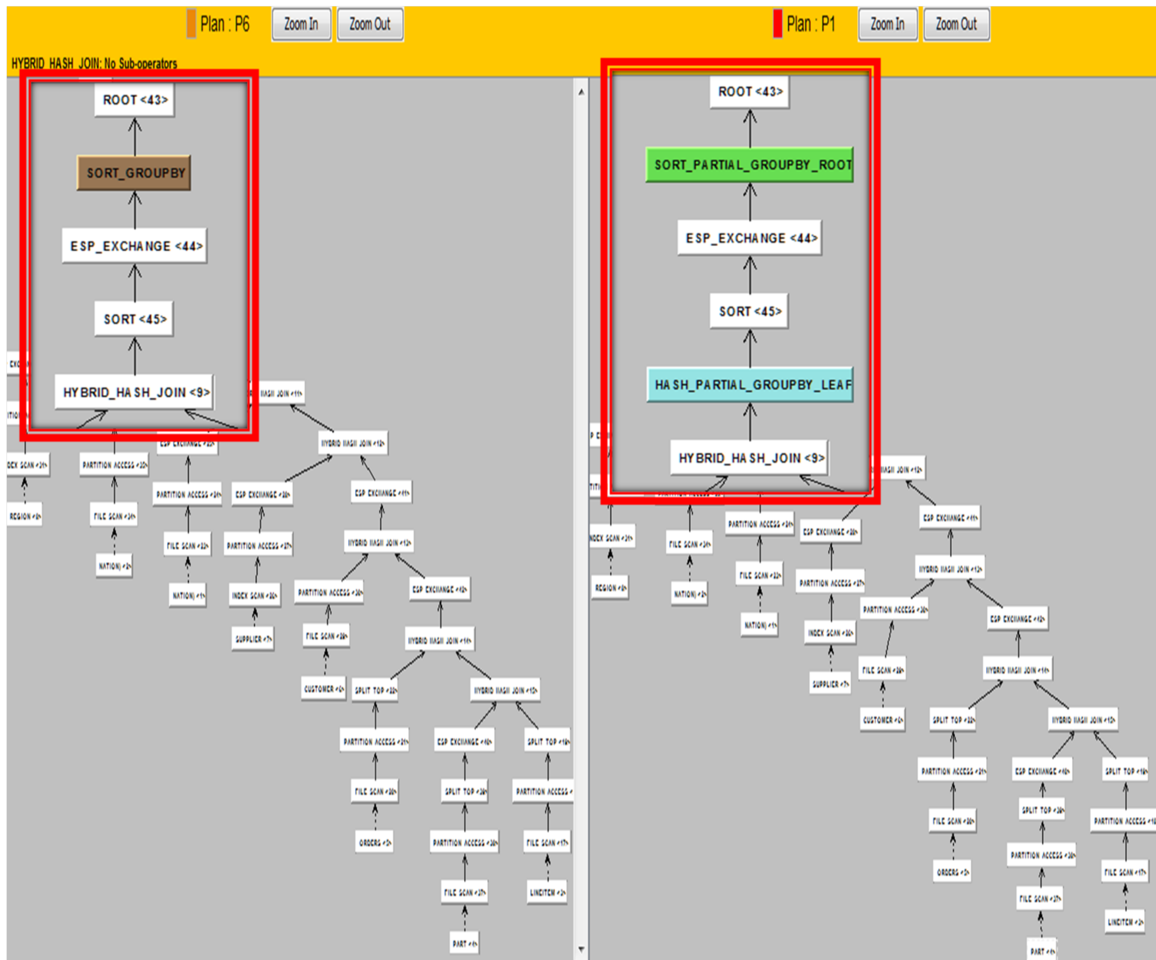

Figure 4.8: Difference in group by operation

difference here is only on operator basis. Parameter level information associated with each operator is not taken into consideration for plan difference. As ESP_EXCHANGE operator can occur anywhere in the plan tree, we have seen that in the plan diagram the differences in the plans were sometimes because the optimizer has chosen to put the EXP_EXCHANGE operator at a different level in the tree. Figure 4.7 highlights the EXP_EXCHANGE operator's position difference in plan trees of two adjacent plans.

Apart from this difference, some of the trees were differing because of the choice of partial group by as opposed to full group by operation. Figure 4.8 highlights the difference in the plan tree because of partial group by operation. It shows that for plan P6 it has chosen to do full group by operation while for plan P1 it is performing partial group

Query Template	OPT A		OPT B		OPT C		OPT D		COM_OPT	
	P	R	P	R	P	R	P	R	P	R
2	23	7	21	10	74	13	3	2	137	15
3	12	3	8	4	25	11	9	5	28	8
4	7	7	4	2	12	4	1	1	15	7
8	62	3	12	3	33	8	14	4	2041	17
9	49	5	3	3	31	1	4	2	52	15
21	16	8	6	3	47	13	5	2	131	27

P: Cardinality of Plan Diagram
R: Cardinality of Reduced Plan Diagram
(Cost Greedy Reduction algorithm with 20% cost increase threshold)

Figure 4.9: Comparison with centralized optimizers

by. Partial grouping is again one of the special feature of COM_OPT. Also, surprisingly plan P6 was found to occur as a footprint pattern within the optimality region of plan P1 thus violating the assumption of PQO.

4.5 Comparison with centralized db optimizers

Figure 4.9 shows the logical comparison of cardinality of plan diagram and reduced plan diagram of COM_OPT with that of other centralized database optimizers which have already been analyzed in past using Picasso [4]. The results for centralized optimizers were taken from [8]. The cardinality of plan diagram for COM_OPT is order of magnitudes higher than that of other optimizers. Also, on reducing the plan diagram using cost greedy reduction (point wise reduction) technique with 20% cost increase threshold, the reduction in number of plans is significant as compared to original plan diagram as we can see the case for Query Template 8 where 2041 plans got reduced to 17 plans. This shows that optimizer could have produced less number of plans without compromising much on the cost of plans. After reduction it does not reaches anorexic level (around 10

plans) for some of the queries, while in case of other optimizers any complex diagram reaches anorexic level at this cost increase threshold. The reason for less reduction rate is violation of plan cost monotonicity across the selectivity space.

The analysis over the diagrams shows that there is a lot of scope in improving the optimizer's design.

Chapter 5

Making construction of futuristic metadata scenarios easier

In previous chapters we have discussed the analysis of behavior of the optimizer COM_OPT. The analysis of the behavior of current optimizer's design was done using the metadata statistics stored. There exists similar applications for testing and analysis of optimizers which uses only metadata statistics for their purpose. But for effective testing the optimizer must be analyzed for various alternative futuristic metadata scenarios. For creating one such metadata scenario, first corresponding data needs to be constructed, then data must be loaded into the database and then we need to collect statistics over the data. The process is shown in Figure 5.1. Time and space overheads involved in this process makes it infeasible to construct various alternative metadata scenarios for testing (shown in Figure 5.2).

For solving this problem recently a tool "The CODD Metadata Processor" was developed

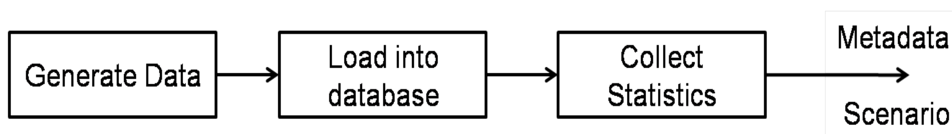

Figure 5.1: Conventional Method for Generating Metadata

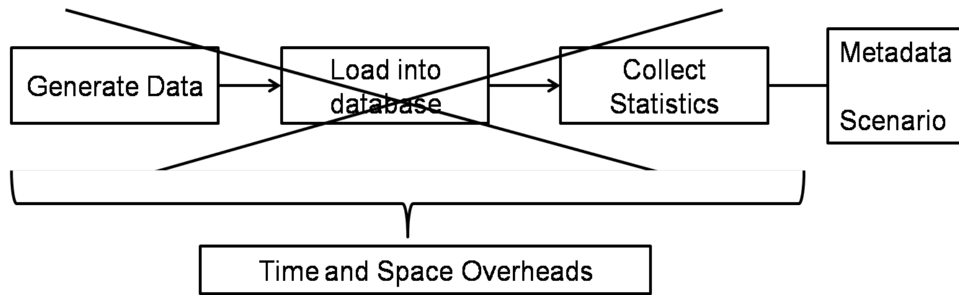

Figure 5.2: Time and Space Overheads in Conventional Process

Figure 5.3: CODD for Constructing Metadata Statistics

which provides graphical interface to users for creating alternative metadata scenarios without the need of corresponding data. Figure 5.3 shows the role of CODD in solving this problem. With the help of this tool alternative metadata scenarios can be constructed in minutes. Distributed database systems are designed to handle large amount of data and therefore must be ideally testing using metadata scenarios corresponding to big data. But creating such scenarios using conventional method is totally infeasible. The tool CODD can be extremely helpful for such databases. Therefore, in this work we have engineered CODD for supporting COM_OPT which corresponds to a distributed database system.

Chapter 6 provides a brief introduction to the tool CODD and rest of the chapters in the thesis describes our contributions towards this tool.

Chapter 6

The CODD Metadata Processor

CODD, is a JAVA based graphical tool that supports the *ab initio* creation of metadata. It provides a graphical user interface through which databases with the desired metadata characteristics can be efficiently simulated without persistently generating and/or storing their contents. This makes the construction of futuristic metadata scenarios feasible and therefore helps in effective testing of optimizers. For construction of various alternative metadata scenarios, CODD provides the following modes of operations:

1. **Metadata Construction:** This mode provides engine specific graphical interface wherein user can input the metadata statistics for the desired scenario. Figure 6.1 shows the overview of utility of construct mode. It also provides a graphical editing interface in the form of graphical histograms to visually alter the attribute-value distributions. It also incorporates a graph based validation algorithm that ensures that the input metadata values are both *legal* (valid range, correct type) and *consistent* (compatible with other metadata values). After validation of inputted metadata statistics, the metadata statistics in the database catalogs are updated to complete the construction of desired scenario. The whole process of metadata input including visual alteration of histograms, validation and catalog updates can be completed in few minutes. In short, this mode lets the user construct various alternative scenarios ranging from *empty* to *Big-Data* (for instance, *yottabyte* [10^{24}]) sized relational tables, *uniform* to *skew* attribute-value distribution, etc. without

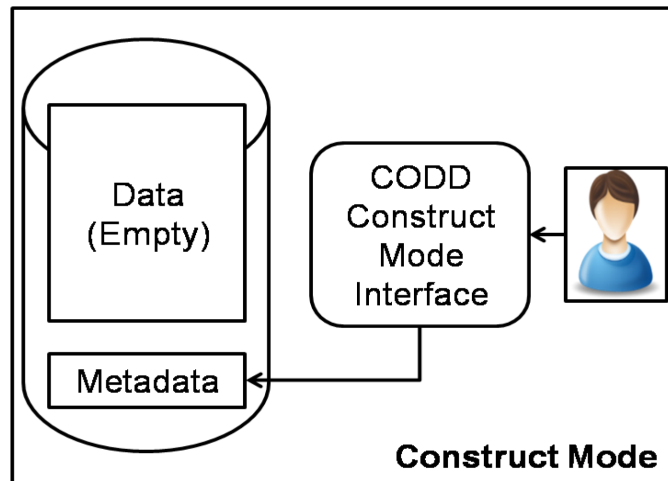

Figure 6.1: Construct Mode

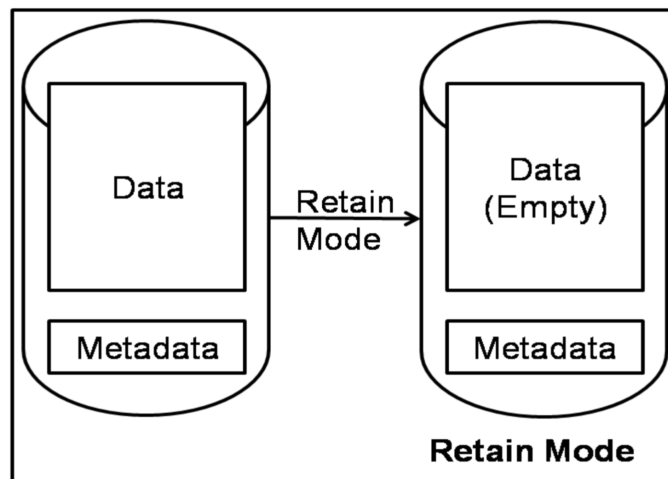

Figure 6.2: Retain Mode

requiring the presence of any prior data instance.

2. **Metadata Retention:** There are scenarios when metadata statistics are required to be generated only from the actual data. In such cases, with retention mode of CODD the raw data can be *dropped* subsequently to get the raw data storage space back without affecting the metadata statistics. This facilitates the testers to temporarily load the real-world database scenarios without incurring storage and maintenance overheads of data during the testing process. Figure 6.2 shows the overview of retain mode.

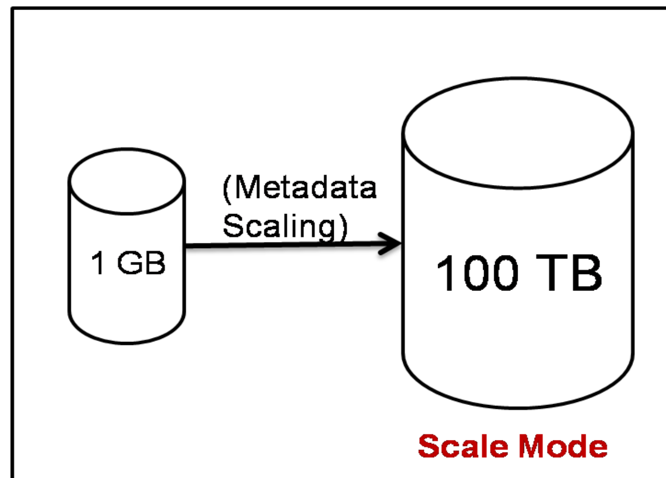

Figure 6.3: Scale Mode

3. **Metadata Scaling:** In practice, testing of database engines is usually done on scaled versions of original databases. CODD provides the facility of scaling the metadata instances as shown in Figure 6.3. It performs follwing two types of scaling over metadata:

   - *Size scaling:* Given a baseline metadata instance and a user-specified scaling factor, it produces the scaled metadata instance. For example, given a metadata instance corresponding to 1GB database and scaling factor 100, CODD produces the scaled metadata instance corresponding to 100GB database as shown in Figure 6.3. But the underlying data (if exists) will remains same as before scaling.
   - *Time Scaling:* Given a query workload, a baseline metadata instance and a user-specified scaling factor, the metadata instance is scaled such that the overall estimated execution time of the query workload is scaled by the user-specified scaling factor.

In a nutshell, CODD is an easy-to-use graphical tool for the automated creation, verification, retention and scaling of database metadata configurations.

6.1 Our Contributions

In this work, we have done following two enhancements to the tool:

- **Architectural Changes:** Redesigned CODD with *3-tier* architecture
- **Database Addition:** We have engineered the tool CODD to support construction of alternative metadata scenarios for distributed database COM_OPT

Chapter 7 describes the new client-server architecture of CODD and Chapter 8 describes the engineering of CODD for supporting COM_OPT.

Chapter 7

Redesigning CODD Architecture

The tool CODD was initially implemented as a 2-tier application as shown in Figure 7.1, with the application running on client system and making connection to a database which can run on same or a different system. All the processing is done by the application on client side. The client side application sends request to database for query processing. In this work, we have converted the architecture of tool CODD to 3-tier architecture where *CODD client, CODD Server* and *databases* forms the three tiers of the application. Rest of the Chapter provides the motivation behind the re-architecture of the tool followed by description of the new design architecture.

7.1 Motivation

Sequential inserts and updates in a database are computationally expensive operations since they require lot of CPU time and memory resources. Thus they needs systems

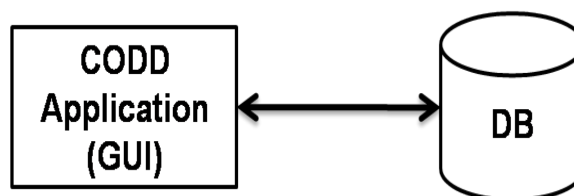

Figure 7.1: Earlier 2-tier CODD Architecture

with high processing capabilities. To increase the performance, the application performing database operations should run on the same system on which database exists. But there exists some non GUI compliant operating systems on which some of the leading commercial database engines run. In such case, the applications with separate user interaction modules and database processing modules are beneficial, and serves both purposes i.e. provides graphical interface to users and improved performance by executing the part of application that interacts with the database servers on the machine on which the database resides. COM_OPT was one of the database systems that runs on a Non-GUI complaint operating system. Thus, before engineering CODD over COM_OPT we have redesigned the tool CODD with 3-tier client-server architecture.

7.2 Architectural Overview

The new architecture of the CODD tool as shown in Figure 7.2 follows a 3-tier architecture, where *CODD client, CODD server and databases* forms the three tiers of the application. With this architecture a client can connect to the server which in turn can connect to a database using JDBC driver depending on the clients request.

The applications requires communication between two different host systems. In JAVA, sockets are the basic communication mechanism for such applications. However, sockets just enables you to send bytes from one machine to the other, and nothing more. They do not provide a way to call methods on other machine. Some application level protocols are required to be implemented for encoding and decoding the messages exchanged between the client and the server applications, which can be error-prone and makes the application complex. JAVA Remote Method Invocation (RMI) [9] is a solution for such problems. It performs the object oriented equivalent of remote procedure calls, with support for direct transfer of serialized JAVA objects. Thus, we have used RMI for client server communication.

Following is the description of the two components of the application.

- **CODD Client:** CODD client provides an interface to the users for inputting or

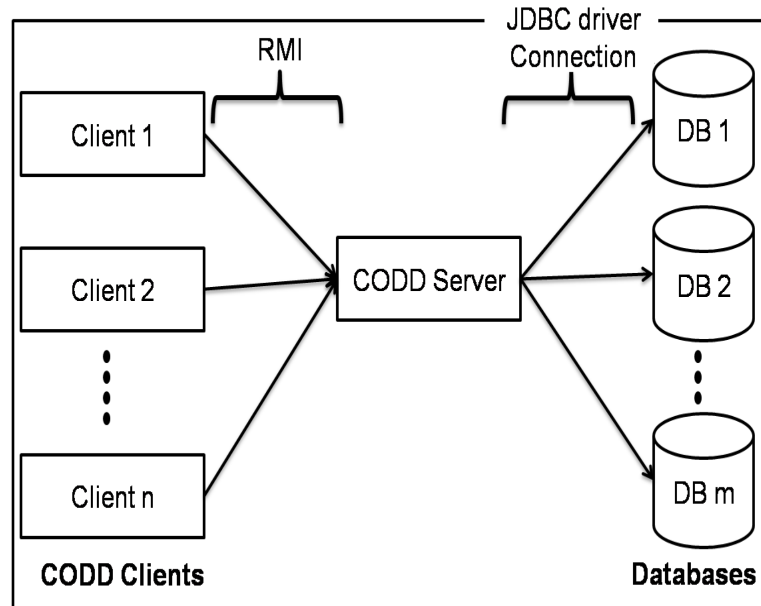

Figure 7.2: CODD 3-tier architecture

modifying metadata statistics of a database. It performs light weight operations on the inputted information like validating the information, optimization of a function, etc. Apart from this, all the query processing tasks are done by server side.

- **CODD Server:** CODD server is implemented for various databases, which implements a common database interface. This common database interface is provided as a stub on client side. RMI registry required for RMI communication with client runs on server side. All the query processing work is done by CODD server. It uses JDBC driver to interact with the database.

Figure 7.3 gives a breif overview of how the 3-tier architecture of CODD is implemented along with its working. On client side there is a client program that provides interface to the users. While at server side there is a server program that performs query processing tasks for all the databases. For invoking the methods on server side, a common interface has been provided which resides on both sides. The whole application now works as

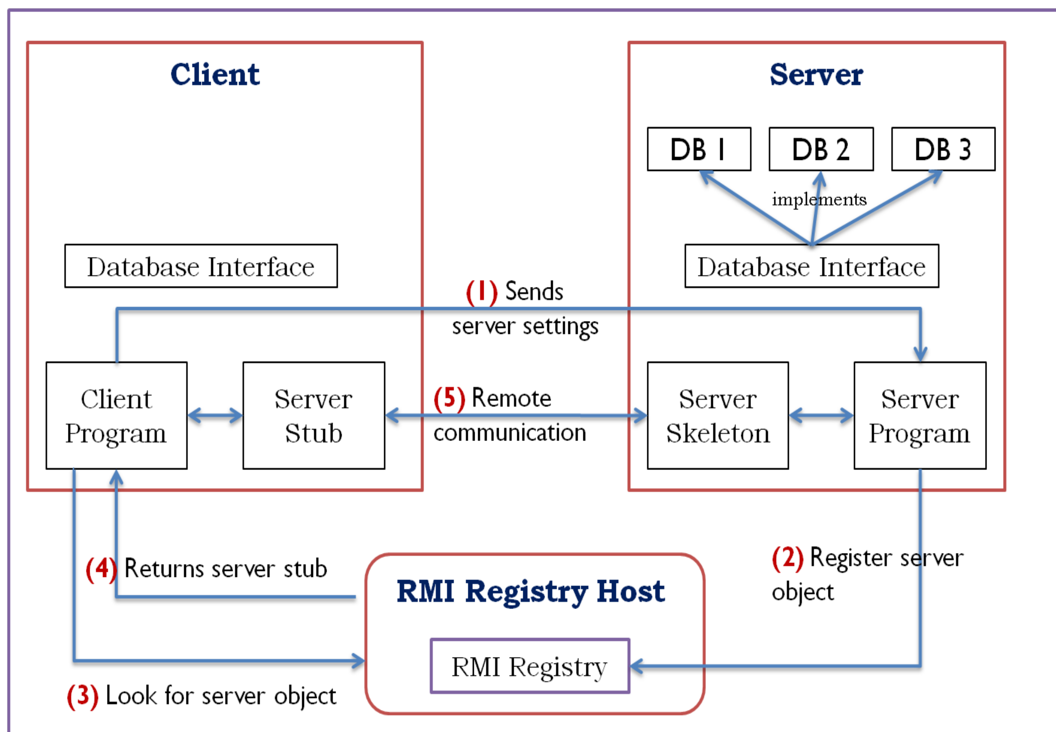

Figure 7.3: Working of CODD 3-tier Architecture

follows: First client sends the database settings to the server. Accordingly server connects to the required database and registers its object in RMI registry. Then for further processing client looks for the server object in RMI registry and gets a server stub. This stub gives an illustration to the client that all methods are present on the same site, but actually it makes remote communication with the server for the method execution.

Chapter 8

Engineering CODD for COM_OPT

CODD is currently available for constructing metadata scenarios for leading commercial centralized databases. In this work, we have engineered the tool CODD to support COM_OPT which is a distributed database system.
Major task in implementation is to extract the schema information and metadata statistics for relations. Since the data in distributed database system is distributed across various nodes, therefore in database system also the schema and data distribution information needs to be stored. In COM_OPT these information's are stored in system metadata tables in highly normalized form. Thus, to get the schema information or metadata statistics, multiple system metadata tables needs to be joined. Also, COM_OPT maintains data distribution statistics for group of columns along with individual columns i.e. it supports multi-column histograms. Thus, we have engineered the tool CODD to support multi-column histograms. In this chapter we will describe the implementation details of various modes of CODD for COM_OPT.

8.1 Metadata Construction

Metadata Construction mode allows user to create metadata statistics without presence of any prior data instance. This section provides the details about implementation of statistics creation and validation in this mode.

Figure 8.1: Construct Mode interface for COM_OPT

8.1.1 Ab-Initio metadata construction

The relations (logical schema) are created first and the statistics are updated using UPDATE statics command to create an entry for the relation and its attributes in HISTOGRAMS and HISTOGRAM_INTERVALS table. Then the metadata construction procedure is started. Figure 8.1 shows the construct mode interface for COM_OPT. The metadata statistics are grouped into two categories: relation level and column level. Index level metadata information (e.g. index level of last statistics update, number of non empty blocks at last statistics update, etc.) is also available with COM_OPT but are stored in system metadata tables which cannot be modified by users. Therefore, index level statistics are excluded from construct mode interface.

Relation level metadata statistics includes relation cardinality. Apart from relation cardinality, COM_OPT stores the RECORD_SIZE (number of bytes in each logical record), BLOCK_SIZE (number of bytes for disk blocks), etc. But these information's are stored in system metadata tables which cannot be modified by users, thus we have not included these in the construct mode interface. Column level metadata statistics for a column or group of columns includes number of distinct values, High Value (highest value in the column/ highest value of each column in a group of columns) and Low Value (lowest value in the column/ lowest value of each column in a group of columns).
For each relation, user inputs the relation level metadata statistics, followed by column level statistics for each column and group of columns for which entry exists in HISTOGRAMS table. After each update (relation level or column level), CODD validates the input metadata statistics and if there is a validation error it reports it to the user with the appropriate error message. For example, if the entered High Value is less than Low Value, then an error message appears asking user to re-enter the correct values.
In COM_OPT, equi-width histograms are created which summarizes the data distribution with a set of buckets. Construct mode interface allows user to input the histogram statistics either manually or from a file. Subsequently, with the Graphical Histogram feature of CODD these values can be viewed graphically and its layout can be visually altered to get the desired data distribution by simply reshaping the bucket boundaries.
After inputting the required metadata statistics, actual construction of metadata instance takes place by updating the database catalogs with the entered metadata values.

8.1.2 Metadata Validation

In metadata construction, user inputs the metadata statistics. Before updating these values to the database catalogs, we need to ensure that each of the input metadata value satisfies the following two types of constraints:

- **Structural Constraints:** Input metadata value must be of *specified type* and it must fall in the *specified range* of values. For example, cardinality of a relation must be of integer type and the value must be greater than or equal to zero.

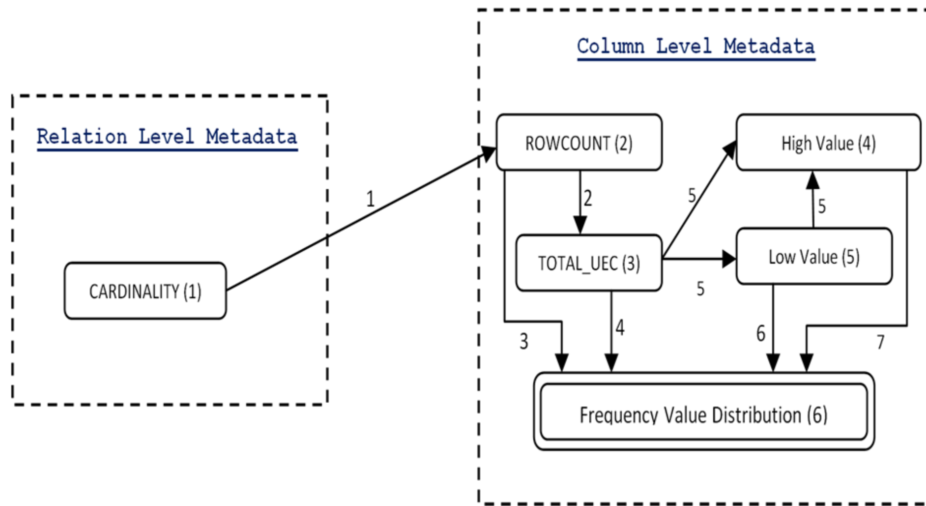

Figure 8.2: Constraint Graph

- **Consistency Constraints:** Input metadata value must be compatible with other metadata values. For example, number of distinct values must be less than or equal to the cardinality of that relation.

The validation process in CODD first constructs a *directed acyclic constraint graph*, which represents all the metadata entities along with its structural and consistency constraints. The details can be found in [3]. Figure 8.2 shows the constructed directed acyclic constraint graph for COM_OPT. The nodes of the graph are populated with the corresponding entries on the construct mode interface. For example, node CARD is populated with cardinality of the relation. One of the structural constraint over this node specifies that the value should be a whole number. The graph edges are added based on the consistency constraints between metadata entities. Table 8.1 lists all the validation constraints.

After constructing the constraint graph G(V,E), CODD runs a *topological sort* on G. The sort provides a linear ordering of the nodes. A sample linear ordering is shown through the numbers associated with the nodes. After getting the input from the user through CODD metadata construct mode interface, the input values are validated by traversing

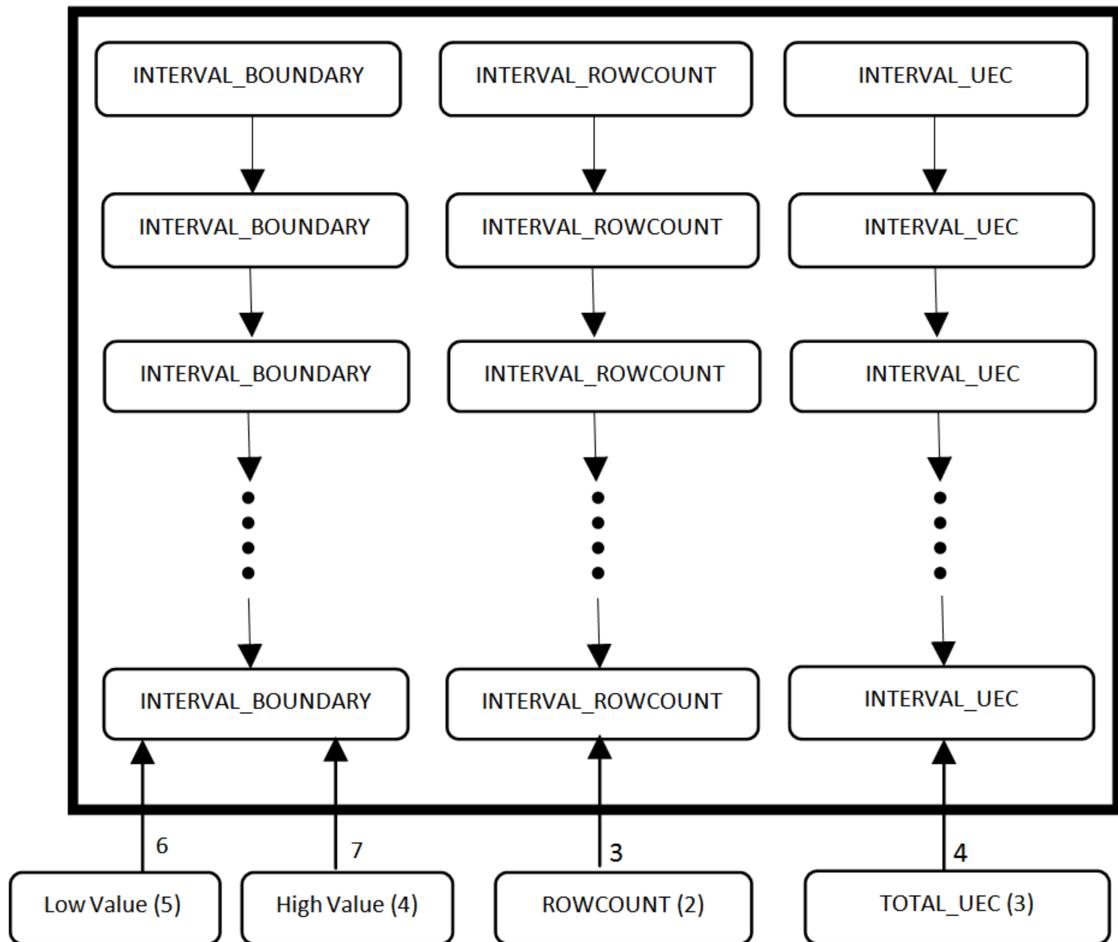

Figure 8.3: Frequency Value Histogram

through the linear ordering and at each node validating the structural and consistency constraints. If any of the constraint is not satisfied then it is reported to the user. After validating all the user input metadata values, they are updated into the database catalogs to complete the metadata construction.

8.1.3 Implementation Details

Here, we present the mechanism by which the input metadata statistics are added or updated into the user metadata tables of COM_OPT.

Constraint	Description
1	The cardinality of column (ROWCOUNT) cannot be greater than the cardinality of its corresponding table (CARDINALITY)
2	The number of distinct values in a column (TOTAL_UEC) cannot be greater than the cardinality of the column (ROWCOUNT)
3	The sum of the values in INTERVAL_ROWCOUNT must be equal to the number of rows in the column (ROWCOUNT)
4	The sum of the values in INTERVAL_UEC must be equal to the number of distinct rows in the column (TOTAL_UEC)
5	High Value is greater than Low Value whenever there are more than three distinct values in the corresponding column. In case of multi-column histograms, the High value and Low Value of each columns must match with the High Value and Low Value of corresponding single column histograms over those columns
6,7	INTERVAL_BOUNDARY values should lie between High Value and Low Value. The last value for INTERVAL_BOUNDARY should be equal to High Value. And the values in INTERVAL_BOUNDARY must be increasing

Table 8.1: Consistency Constraints

HISTOGRAMS table, which is stored in same schema in which relation is stored, contains the relation level statistics. Column level metadata statistics are stored in HISTOGRAMS and HISTOGRAM_INTERVALS table. HISTOGRAMS table stores number of unique rows in the table for the column, Low Value and High Value for the column. The data distribution for columns is stored in HISTOGRAM_INTERVALS table. COM_OPT supports direct INSERT and UPDATE commands over these table. CODD performs metadata construction in following steps (also shown in Figure 8.4):

- Create relation schemas.
- Then populate the catalog tables by executing UPDATE STATISTICS command to make empty entries corresponding to the relations in them.
- Then it updates them with the user provided input metadata statistics or inserts new entries in the catalog tables using INSERT command.

For tables without user-defined clustering key, COM_OPT automatically adds an additional column SYSKEY(system defined clustering key) to the relation, which then acts as

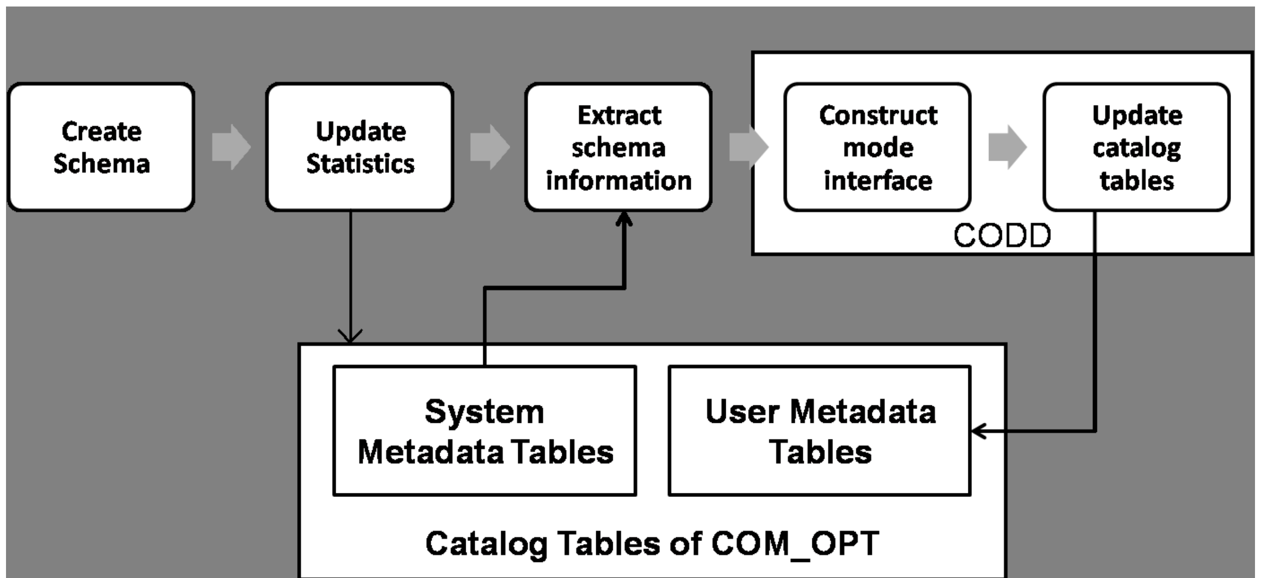

Figure 8.4: Construct Mode Implementation Overview

a clustering key. Whenever a record is inserted in the relation, the system automatically generates a value for the SYSKEY column. During metadata construction, the statistics for SYSKEY column are handled explicitly by the tool. A random value is generated as a LOW_VALUE for this column. HIGH_VALUE is obtained by adding relation cardinality to LOW_VALUE. Rest of the column distribution statistics for SYSKEY column are then automatically generated by the tool accordingly.

Using the construct mode interface for COM_OPT, we were able to construct metadata instance with relation cardinality of 10^{18}, which is the maximum supported cardinality by the database system.

8.2 Metadata Retention

Metadata retention mode of CODD drops the data of specified relations from database and reclaims the raw data space back without affecting the metadata statistics of those relations. Rest of the section provides the implementation details of this mode for COM_OPT.

8.2.1 Implementation Details

Referential integrity constraint does not allow a foreign key relation tuple to be present without its corresponding primary key relation tuple. So if data of a relation has to be dropped, all its dependent relations must be dropped before dropping the relation. For example, in TPC-H benchmark, dropping REGION requires NATION to be dropped first as NATION refers to REGION by N_REGIONKEY. Thus, given a set of relations to drop the data, CODD first finds the dependent relations and adds them to the drop list. It then performs following steps for carrying out Metadata Retention:-

1. Collection of following details about Foreign Key's for all relations:
    - Foreign key name
    - Foreign key columns
    - Corresponding primary key relations
    - Corresponding primary key columns
2. Drop the Foreign key's over the relation's schema
3. Delete the data from the relations using DELETE command
4. Modify the relation's schema by recreating the Foreign key constraints over them using above collected statistics

In COM_OPT, the metadata statistics for a relation are updated only when user issues an UPDATE STATISTICS command for the relation, therefore the metadata statistics are retained after deleting the data from relations.

8.3 Metadata Scaling

CODD provides two types of metadata scaling: *Size Scaling* and *Time Scaling*. Implementation of the two scaling models of CODD for COM_OPT are explained in this section.

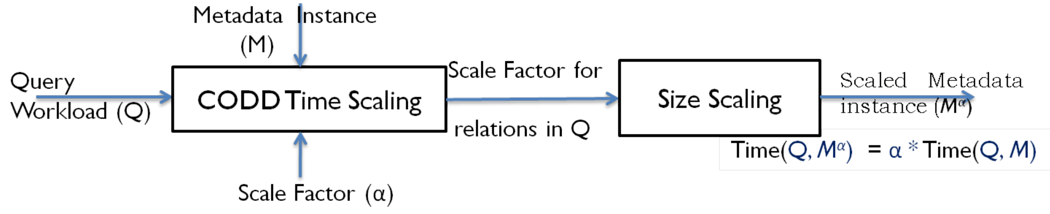

Figure 8.5: Time Scaling Overview

8.3.1 Size-based Scaling

For a given baseline metadata instance and a user specified scaling factor α, the scaling mode of CODD produces a scaled metadata instance such that the space (size in Bytes) occupied by the scaled metadata instance is α times of the baseline metadata instance.

Cardinality scaling on metadata is implemented at relation and column level. At the relation level, we scale the cardinality of relations by α. At the column level, cardinality scaling is implemented differently for key columns and non-key columns. For key columns domain scaling (data distribution domain is scaled) is implemented and the distinct count is scaled by α. For non key columns, we keep the relative frequency distribution same as baseline metadata. In case of multicolumn histograms, distinct count is scaled by α if the columns contains a key column and values corresponding to the key column is scaled in the data distribution.

8.3.2 Time-based Scaling

For a given baseline metadata $\mathcal{M}$, query workload $\mathcal{Q}$ and a user specified scaling factor α, the Time Scaling mode of CODD produces a scaled metadata instance $\mathcal{M}^{\alpha}$ such that the total optimizer's estimated cost (measure of time) of executing $\mathcal{Q}$ on the scaled version is α times the total optimizer's estimated cost of executing $\mathcal{Q}$ on the baseline metadata instance.

The Time Scaling algorithmic details of CODD can be found in [3]. Figure 8.5 shows

Query/ Cost	Cost before scaling	Cost after scaling	Obtained scaling
Q1	61.01	182.43	2.99
Q14	38.84	110.88	2.85
Q17	134.32	430.91	3.20

Table 8.2: Cost of queries before and after scaling

the overview of the time scaling. Following steps are performed for time scaling a given query workload:

1. Read the metadata statistics corresponding to the relations involved in the query workload.
2. Extracts the execution plan for the input queries.
3. Performs time scaling to get scaling factors for individual relations.
4. Performs size based scaling over the relations involved in the query workload with the scaling factors obtained for them.

Example: Consider a 1GB TPC-H workload consisting of queries Q1, Q14 and Q17 that operates on relations PART and LINEITEM with probabilities (0.5, 0.48, 0.02) and scaling factor of 3. With time scaling of this query workload on COM_OPT, we have obtained a scaling factor (1,3) for the two relations respectively. Relation LINEITEM was scaled three times and the cost of queries before scaling and after scaling is given in Table 8.2. The cost of query workload was scaled by the scaling factor of 3.09.

Chapter 9

Conclusions

- We have engineered the tool Picasso to support a query optimizer of a distributed database COM_OPT. The analysis of the behavior of COM_OPT using optimizer diagrams produced by Picasso has shown deviation from the expected behavior by a optimizer. The optimizer makes fine grained plan choices across the selectivity space. Plan diagrams shows that the assumptions of PQO does not holds here. We have also shown PCM violations in cost diagrams. This shows that there is a lot of scope for improving the optimizer's design.

- COM_OPT being a distributed database is used for dealing with huge amount of data. For making the futuristic testing using alternative metadata scenarios, corresponding to large data, feasible and effective for COM_OPT we have engineered the tool CODD to support COM_OPT.

- For saving the performance degradation of CODD for the databases which runs on Non-GUI complient operating systems due to network overheads in query processing we have also redesigned the tool CODD with 3-tier client-server architecture. The new architecture is simpler and scalable as well.

Appendix A

Following are the DDL's of the TPCH relations which we have used for producing Picasso Diagrams.

- REGION

  CREATE TABLE REGION

  (R_REGIONKEY INTEGER NOT NULL, R_NAME CHAR(25) NOT NULL, R_COMMENT VARCHAR(152));

- NATION

  CREATE TABLE NATION

  (N_NATIONKEY INTEGER NOT NULL, N_NAME CHAR(25) NOT NULL, N_REGIONKEY INTEGER NOT NULL,

  N_COMMENT VARCHAR(152));

- SUPPLIER

  CREATE TABLE SUPPLIER

  (S_SUPPKEY INT PRIMARY KEY NOT NULL , S_NAME CHAR(25) , S_ADDRESS VARCHAR(40)

  , S_NATIONKEY INT , S_PHONE CHAR(15), S_ACCTBAL DECIMAL(15, 2), S_COMMENT VARCHAR(101)

  );

- CUSTOMER

  CREATE TABLE CUSTOMER

(C_CUSTKEY INTEGER PRIMARY KEY NOT NULL, C_NAME VARCHAR(25) NOT NULL,

C_ADDRESS VARCHAR(40) NOT NULL, C_NATIONKEY INTEGER NOT NULL,

C_PHONE CHAR(15) NOT NULL, C_ACCTBAL DECIMAL(15,2) NOT NULL,

C_MKTSEGMENT CHAR(10) NOT NULL, C_COMMENT VARCHAR(117) NOT NULL)

RANGE PARTITION BY (C_CUSTKEY)

(ADD FIRST KEY 1 LOCATION $DATA1 NAME PARTNC1 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 50000 LOCATION $DATA1 NAME PARTNC2 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 100000 LOCATION $DATA2 NAME PARTNC3 EXTENT (4096, 4096) MAXEXTENTS 512);

- PART

CREATE TABLE PART

(P_PARTKEY INTEGER PRIMARY KEY NOT NULL, P_NAME VARCHAR(55) NOT NULL,

P_MFGR CHAR(25) NOT NULL, P_BRAND CHAR(10) NOT NULL,

P_TYPE VARCHAR(25) NOT NULL, P_SIZE INTEGER NOT NULL,

P_CONTAINER CHAR(10) NOT NULL, P_RETAILPRICE DECIMAL(15,2) NOT NULL,

P_COMMENT VARCHAR(23) NOT NULL)

RANGE PARTITION BY (P_PARTKEY)

(ADD FIRST KEY 1 LOCATION $DATA1 NAME PARTN1 EXTENT (4096, 4096) MAXEXTENTS 512

, ADD FIRST KEY 50000 LOCATION $DATA1 NAME PARTN2 EXTENT (4096, 4096) MAXEXTENTS 512

, ADD FIRST KEY 100000 LOCATION $DATA2 NAME PARTN3 EXTENT (4096, 4096) MAXEXTENTS 512

, ADD FIRST KEY 150000 LOCATION $DATA2 NAME PARTN4 EXTENT (4096, 4096) MAXEXTENTS 512);

- PARTSUPP

CREATE TABLE PARTSUPP

(PS_PARTKEY INTEGER NOT NULL, PS_SUPPKEY INTEGER NOT NULL,

PS_AVAILQTY INTEGER NOT NULL, PS_SUPPLYCOST DECIMAL(15,2) NOT NULL,

PS_COMMENT VARCHAR(199) NOT NULL, PRIMARY KEY(PS_PARTKEY, PS_SUPPKEY))

RANGE PARTITION BY (PS_PARTKEY)

(ADD FIRST KEY (1) LOCATION $DATA1 NAME PARTNPS1 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (20000) LOCATION $DATA1 NAME PARTNPS2 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (30000) LOCATION $DATA1 NAME PARTNPS21 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (40000) LOCATION $DATA1 NAME PARTNPS3 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (50000) LOCATION $DATA1 NAME PARTNPS31 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (60000) LOCATION $DATA1 NAME PARTNPS4 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (70000) LOCATION $DATA1 NAME PARTNPS41 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (80000) LOCATION $DATA1 NAME PARTNPS5 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (90000) LOCATION $DATA1 NAME PARTNPS51 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (100000) LOCATION $DATA1 NAME PARTNPS6 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (110000) LOCATION $DATA1 NAME PARTNPS61 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (120000) LOCATION $DATA2 NAME PARTNPS7 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (130000) LOCATION $DATA2 NAME PARTNPS71 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (14000) LOCATION $DATA2 NAME PARTNPS8 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (15000) LOCATION $DATA2 NAME PARTNPS81 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (160000) LOCATION $DATA3 NAME PARTNPS9 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (170000) LOCATION $DATA3 NAME PARTNPS10 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY (180000) LOCATION $DATA4 NAME PARTNPS11 EXTENT (4096, 4096) MAXEXTENTS 512);

- ORDERS

CREATE TABLE ORDERS

(O_ORDERKEY INTEGER PRIMARY KEY NOT NULL, O_CUSTKEY INTEGER NOT NULL,

O_ORDERSTATUS CHAR(1) NOT NULL, O_TOTALPRICE DECIMAL(15,2) NOT NULL,

O_ORDERDATE DATE NOT NULL, O_ORDERPRIORITY CHAR(15) NOT NULL,

O_CLERK CHAR(15) NOT NULL, O_SHIPPRIORITY INTEGER NOT NULL,

O_COMMENT VARCHAR(79) NOT NULL)

RANGE PARTITION BY (O_ORDERKEY) (

ADD FIRST KEY 1 LOCATION $DATA1 NAME PARTNO1 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 500000 LOCATION $DATA1 NAME PARTNO2 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1000000 LOCATION $DATA1 NAME PARTNO3 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1500000 LOCATION $DATA1 NAME PARTNO4 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2000000 LOCATION $DATA1 NAME PARTNO5 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2500000 LOCATION $DATA2 NAME PARTNO6 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3000000 LOCATION $DATA2 NAME PARTNO7 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3500000 LOCATION $DATA2 NAME PARTNO8 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4000000 LOCATION $DATA3 NAME PARTNO9 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4500000 LOCATION $DATA3 NAME PARTNO10 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5000000 LOCATION $DATA3 NAME PARTNO11 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5500000 LOCATION $DATA3 NAME PARTNO12 EXTENT (4096, 4096) MAXEXTENTS 512);

- LINEITEM

  CREATE TABLE LINEITEM

  (L_ORDERKEY INTEGER NOT NULL, L_PARTKEY INTEGER NOT NULL,

  L_SUPPKEY INTEGER NOT NULL, L_LINENUMBER INTEGER NOT NULL,

  L_QUANTITY DECIMAL(15,2) NOT NULL, L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL,

  L_DISCOUNT DECIMAL(15,2) NOT NULL, L_TAX DECIMAL(15,2) NOT NULL,

  L_RETURNFLAG CHAR(1) NOT NULL, L_LINESTATUS CHAR(1) NOT NULL,

  L_SHIPDATE DATE NOT NULL, L_COMMITDATE DATE NOT NULL,

  L_RECEIPTDATE DATE NOT NULL, L_SHIPINSTRUCT CHAR(25) NOT NULL,

  L_SHIPMODE CHAR(10) NOT NULL, L_COMMENT VARCHAR(44) NOT NULL,

  PRIMARY KEY (L_ORDERKEY, L_LINENUMBER))

  RANGE PARTITION BY (l_ORDERKEY) (

  ADD FIRST KEY 1 LOCATION $DATA1 NAME PARTNL1 EXTENT (4096, 4096) MAXEXTENTS

512,

ADD FIRST KEY 100000 LOCATION $DATA1 NAME PARTNL2 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 200000 LOCATION $DATA1 NAME PARTNL3 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 300000 LOCATION $DATA1 NAME PARTNL4 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 400000 LOCATION $DATA1 NAME PARTNL5 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 500000 LOCATION $DATA1 NAME PARTNL6 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 600000 LOCATION $DATA1 NAME PARTNL7 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 700000 LOCATION $DATA1 NAME PARTNL8 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 800000 LOCATION $DATA1 NAME PARTNL9 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 900000 LOCATION $DATA1 NAME PARTNL10 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1000000 LOCATION $DATA2 NAME PARTNL11 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1100000 LOCATION $DATA2 NAME PARTNL12 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1200000 LOCATION $DATA2 NAME PARTNL13 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1300000 LOCATION $DATA2 NAME PARTNL14 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1400000 LOCATION $DATA2 NAME PARTNL15 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1500000 LOCATION $DATA2 NAME PARTNL16 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1600000 LOCATION $DATA2 NAME PARTNL17 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1700000 LOCATION $DATA2 NAME PARTNL18 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1800000 LOCATION $DATA2 NAME PARTNL19 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 1900000 LOCATION $DATA2 NAME PARTNL20 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2000000 LOCATION $DATA3 NAME PARTNL21 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2100000 LOCATION $DATA3 NAME PARTNL22 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2200000 LOCATION $DATA3 NAME PARTNL23 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2300000 LOCATION $DATA3 NAME PARTNL24 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2400000 LOCATION $DATA3 NAME PARTNL25 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2500000 LOCATION $DATA3 NAME PARTNL26 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2600000 LOCATION $DATA3 NAME PARTNL27 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2700000 LOCATION $DATA3 NAME PARTNL28 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2800000 LOCATION $DATA3 NAME PARTNL29 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 2900000 LOCATION $DATA3 NAME PARTNL30 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3000000 LOCATION $DATA3 NAME PARTNL31 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3100000 LOCATION $DATA4 NAME PARTNL32 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3200000 LOCATION $DATA4 NAME PARTNL33 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3300000 LOCATION $DATA4 NAME PARTNL34 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3400000 LOCATION $DATA4 NAME PARTNL35 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3500000 LOCATION $DATA4 NAME PARTNL36 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3600000 LOCATION $DATA4 NAME PARTNL37 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3700000 LOCATION $DATA4 NAME PARTNL38 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3800000 LOCATION $DATA4 NAME PARTNL39 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 3900000 LOCATION $DATA4 NAME PARTNL40 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4000000 LOCATION $DATA4 NAME PARTNL41 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4100000 LOCATION $DATA4 NAME PARTNL42 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4200000 LOCATION $DATA4 NAME PARTNL43 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4300000 LOCATION $DATA4 NAME PARTNL44 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4400000 LOCATION $DATA4 NAME PARTNL45 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4500000 LOCATION $DATA4 NAME PARTNL46 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4600000 LOCATION $DATA4 NAME PARTNL47 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4700000 LOCATION $DATA4 NAME PARTNL48 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4800000 LOCATION $DATA4 NAME PARTNL49 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 4900000 LOCATION $DATA4 NAME PARTNL50 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5000000 LOCATION $DATA4 NAME PARTNL51 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5100000 LOCATION $DATA4 NAME PARTNL52 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5200000 LOCATION $DATA4 NAME PARTNL53 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5300000 LOCATION $DATA4 NAME PARTNL54 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5400000 LOCATION $DATA4 NAME PARTNL55 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5500000 LOCATION $DATA4 NAME PARTNL56 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5600000 LOCATION $DATA4 NAME PARTNL57 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5700000 LOCATION $DATA4 NAME PARTNL58 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5800000 LOCATION $DATA4 NAME PARTNL59 EXTENT (4096, 4096) MAXEXTENTS 512,

ADD FIRST KEY 5900000 LOCATION $DATA4 NAME PARTNL60 EXTENT (4096, 4096) MAXEXTENTS 512);

References

[1] Harish D, Pooja N. Darera and Jayant R. Haritsa, "On the production of anorexic plan diagrams", *Proc. of VLDB 2007*

[2] Y. Ioannidis, R. Ng, K. Shim and T. Sellis, "Parametric Query Optimization"', *Proc of VLDB 1992*

[3] I. Nilavalagan, "The CODD Metadata Processor", M.E. Thesis. DSL/SERC, Indian Instt. of Science (2012)

[4] N. Reddy and Jayant R. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers", *Proc. of VLDB 2005*

[5] R. Trivedi, I. Nilavalagan and J. Haritsa, "CODD: COnstructing Dataless Databases", *Proc of DB Test 2012*

[6] *dsl.serc.iisc.ernet.in/projects/CODD*

[7] *http://en.wikipedia.org/wiki/Gini_coefficient*

[8] *dsl.serc.iisc.ernet.in/projects/PICASSO*

[9] *http://docs.oracle.com/javase/tutorial/rmi/*

[10] *www.tpc.org/tpch*