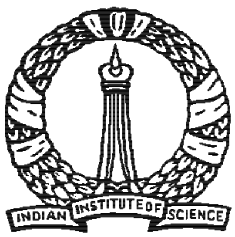
Picasso: Design and Implementation of a Query Optimizer Analyzer

A Project Report
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in
Internet Science and Engineering

by

Mohammed Aslam



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

July 2006

To

My Family

for the unconditional support and trust

Acknowledgments

First of all I would like to thank my guide Prof. Jayant R. Haritsa for his encouragement and support. I had learned a lot under his supervision and he helped me improve in more ways than one.

I am deeply indebted to my family who have been a constant source of support and inspiration for me. I would also like to thank my fellow DSL'ites Akshat, Gopal, Sandeep and Prateem for their cheerful and helpful company which made working in lab a pleasant experience. Last, but not least, I thank my friends and relatives who have in some or the other way helped me in completing this work.

Abstract

Picasso[5] is a tool developed in DSL which produces *plan diagrams*, *plan-cost diagrams* and *plan-cardinality diagrams* for relational query optimizers collectively called as *Picasso diagrams*. *Plan diagram* is a pictorial enumeration of the execution plan choices of a database query optimizer. *Plan-cost diagram* is similar to *plan diagram*, but shows the estimated cost of executing the query over the selectivity space of the query. *Plan-cardinality diagram* is similar to *plan-cost diagram* except that it shows the cardinality of result set instead of cost of execution.

Picasso code was released to the public which has its origin at Plastic[4] project. But the code proved to have become overly complex due to its evolution from a different project. Therefore the tool is re-architected and re-designed for better functionality, performance, features and extensibility. The architecture, design, implementation issues and features of the *Picasso* tool will be discussed in the rest of the report.

Contents

List of Figures

Chapter 1

Introduction

Database Management Systems support declarative access to data rather than procedural access. Users specify only the 'what' part of data access and not the 'how' part. This implies that DBMS systems has to come up with a strategy to access data which produces the desired output. One important problem in this regard is the abundance of strategies, called plans, and orders of magnitude performance difference between best and worst plans, which makes selecting the right plan a very important task. This is even more critical in recent times due to the high degree of query complexity characterizing current data warehousing and mining applications, as shown by TPCH benchmark[24]. An example of a query execution plan is shown in Figure 1.2.

Query optimization is a hard problem[1]. Attempts to build query optimizers based on rules or direct synthesis have not produced any usable results so far. While commercial query optimizers each have their own proprietary methods to identify the best plan, the de-facto standard underlying strategy is based on the classical System R optimizer paper[2]. The proposed method is : Given a user query, apply a variety of heuristics to restrict the combinatorially large search space of plan alternatives to a manageable size; estimate, with a cost model and a dynamic-programming-based processing algorithm, the efficiency of each of these candidate plans; finally, choose the plan with the lowest estimated cost.

Query optimization using this cost-based approach is expensive and takes fair amount of

computational resources even for medium sized queries[3]. Therefore understanding and characterizing query optimizers to improve performance is very important.

The de-facto query language for relational database is SQL. SQL stands for structured query language.Commercial database systems many a time use a dialect of SQL with slight syntactic variations. An example SQL query is given in Figure 1.

```
select l_shipmode
from orders, lineitem
where o_orderkey = l_orderkey
        and o_totalprice <= 100
        and l_quantity <= 20
group by l_shipmode
order by l_shipmode
```

Figure 1.1: Sample SQL Query

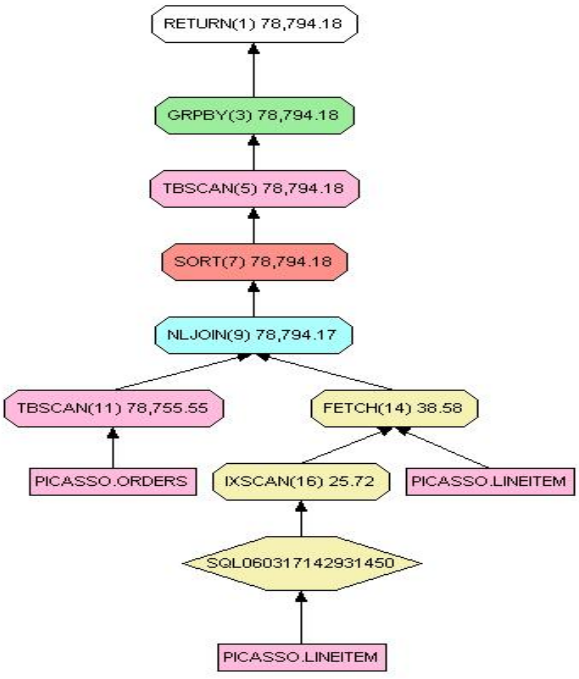


Figure 1.2: Execution Plan

1.1 The Picasso Tool

Even though query optimizer is an important part of any relational database that supports declarative query language like SQL, there is a surprising lack of tools to characterize, evaluate or understand the behavior of cost-based query optimizers. Picasso [6] is a tool developed in DSL [10] which can be used to analyze and characterize relational query optimizers.

SQL queries with range predicates in it tend to produce numerous optimal plans depending on the constants involved in the predicate. Given a relational query optimizer and query with range predicates Picasso can generate a variety of diagrams that throws light into how the optimizer is functioning. The diagrams include plan diagrams, reduced plan diagrams, plan-cost diagrams and plan-cardinality diagrams collectively called *Picasso diagrams*.

Plan Diagram

Plan diagram is a pictorial enumeration of the execution plan choices of a database query optimizers. Plan diagram shows the regions where a plan is optimal according to the chosen optimizer. It also shows the coverage of plans over the selectivity space. An example is shown in Figure 1.3.

Plan-cost and Plan-cardinality diagram

Plan-cost diagram is similar to plan diagram but shows the estimated cost of executing the query over the selectivity space of the query, along with plans, as a third dimension. These diagrams helps in understanding how cost varies within a plan and across plans for the same query over the selectivity space. Plan-cardinality diagram is similar to plan-cost diagram except that it shows the cardinality of result set instead of cost of execution. Examples of plan-cost and plan-cardinality diagrams are shown in Figure 1.4 and 1.5.

Execution Plan-cost and Execution Plan-cardinality diagram

Execution Plan-cost diagram is similar to Plan-cost diagram, but instead of using the estimated cost as reported by the optimizer, the query is executed and time taken is considered as the

cost of the query. The comparison between Execution Plan-cost diagram and Plan-cost diagram shows the quality of the cost model of optimizer.

Execution Plan-cardinality diagram is similar to Plan-cardinality diagram, but instead of using the estimated cardinality as reported by the optimizer, the query is executed and number of tuples is counted for the actual cardinality of result. The comparison between Execution Plan-cardinality diagram and Plan-cardinality diagram shows the quality of the statistics and estimation of the optimizer.

These example diagrams are generated for Query 7 of the TPC-H [24] benchmark, with selectivity variations on the *orders* and *customer* relations.

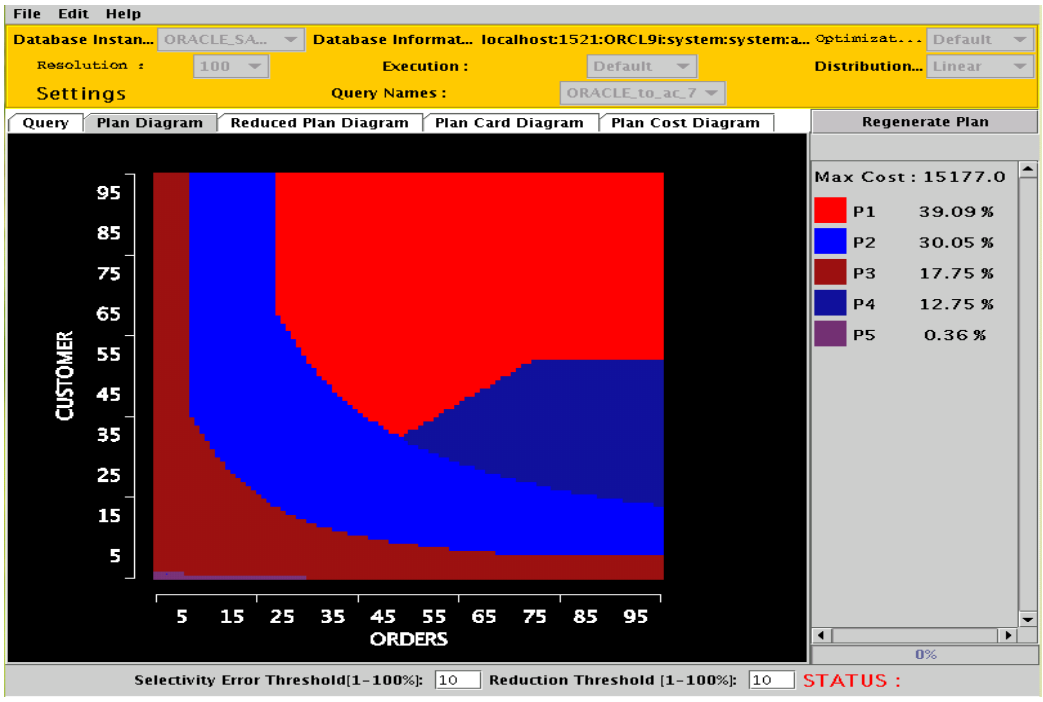


Figure 1.3: Plan Diagram (Query 11, DB2)

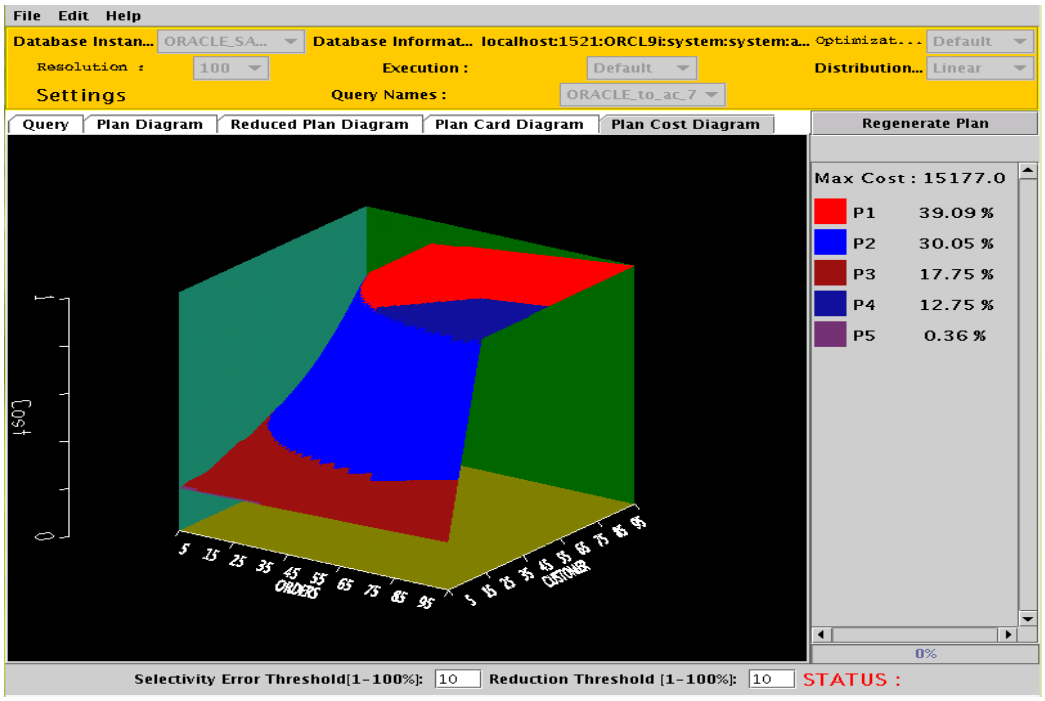


Figure 1.4: Plan-Cost Diagram (Query 11, DB2)

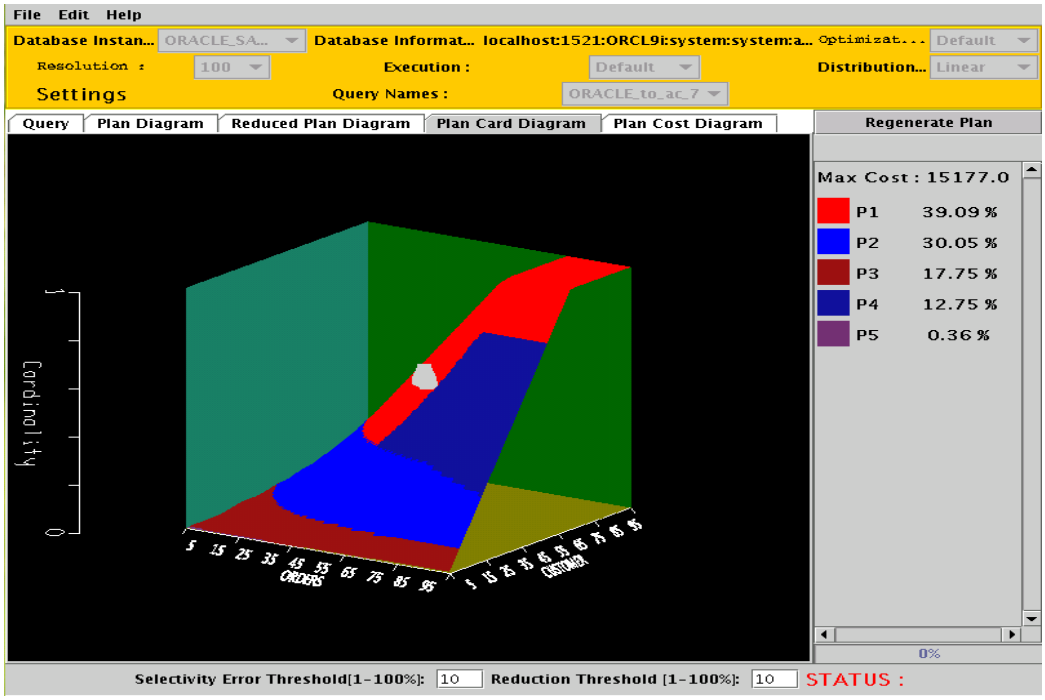


Figure 1.5: Plan-Cardinality Diagram (Query 11, DB2)

Chapter 2

Prior Work

DSL has produced a tool called Plastic[12] which helps in plan re-use using query clustering based on a feature vector. During the initial stages of Picasso project [11], the Picasso tool is created from Plastic code base. The tool was used to generate various Picasso diagrams. Due to its heritage and evolution, the code base of Picasso became unmanageably complex and adding more features became more and more difficult. Some of the requirements such as remote visualization, execution plan diagrams, exponential distribution of selectivity and N dimensional diagrams could not be implemented due to architectural as well as design problems. Therefore to further assist in the analysis of query optimizers, a re-architecting and re-designing was carried out and the new Picasso tool is implemented from scratch.

Chapter 3

Architecture

Picasso follows a 3-tier architecture where *Picasso client*, *Picasso server* and *databases* form the three tiers. Multiple clients can connect to the server which can in turn make use of different databases. This new network centric architecture is depicted in figure 3.1.

The client connects to the server through network and makes requests to the server for retrieving different kinds of information which will be explained in detail later. Server connects to the database, generates different *Picasso diagrams* and stores the data persistently on the database itself.

Generating a *Picasso diagram* is a computationally expensive operation requiring lot of CPU time and memory resources. The fundamental advantage of the client server architecture is that users can use the client on their desktops with very little resources, while the Picasso server which may run on high end dedicated machines can make use of the resources to run at reasonable speed. The databases can be either on the same machine as the Picasso server or can be on another machine on the same network. Having the database on the same machine as Picasso server is slightly faster since lot of network overheads can be avoided.

Server

Picasso server is composed of a *Query Analyzer*, *Query Generator*, *Diagram Generator*, *Selectivity Estimator*, *Database Interface* and *Client Interface* modules. Integrated Client and Server component architecture diagram is shown in Figure 3.2.

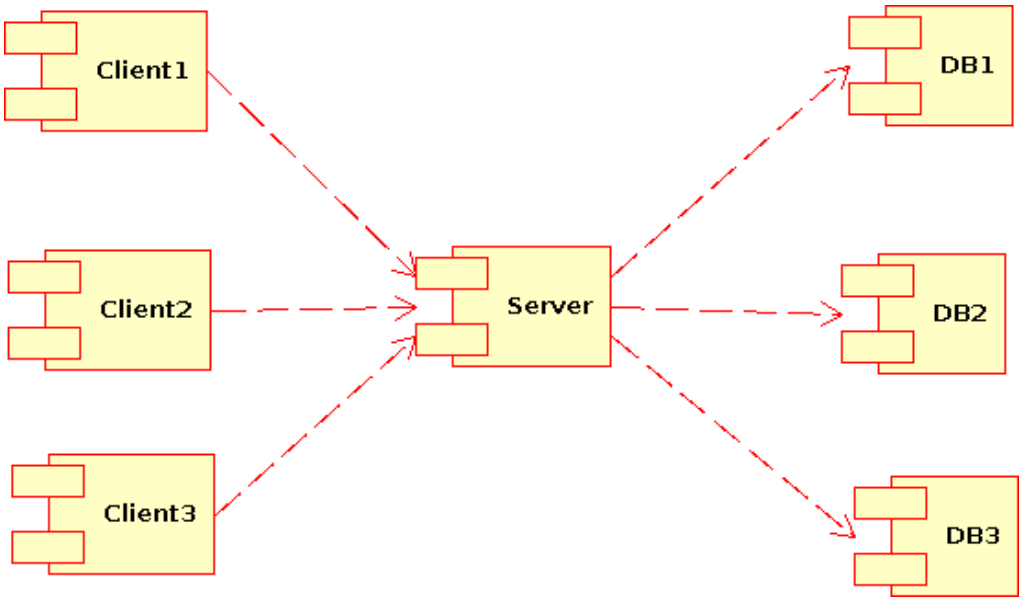


Figure 3.1: System Architecture

- *Query Analyzer* module parses the query template and extracts constant range predicates marked by the format *:num*. It then identifies the relation name and schema name associated with the attribute in the predicate. It also validates the query for correctness. Validation includes checking for duplication of selected attributes or the selection of attributes from the same relation.

- *Query Generator* module generates queries for each point in the selectivity space for the given resolution. The points on selectivity space can be either uniformly or exponentially distributed. It uses the selectivity estimator module to generate constants corresponding to the selectivity of each points. These cached constant values are used to generate the query from query template.

- *Selectivity Estimator* reads the statistics collected in the database such as histograms. Then it generates a constant which when applied on the selected predicate results in the specified selectivity.

- *Diagram Generator* module actually generates the *Picasso diagrams*. It uses Query Generator module to generate queries and fires these queries using *Database Interface* module

to retrieve the optimal plan, estimated cost and estimated cardinality according to the optimizer. All these information including plan trees, optimizer selectivity, Picasso selectivity and constants used are stored persistently to the database periodically.

- *Database Interface* module abstracts away communication to the relational databases. Different databases has slightly varying SQL syntax, different types of histograms and different plan representation. This modules abstracts such inconsistencies under a common interface.
- *Client Interface* module abstracts the communication to the client. Currently Java object serialization mechanism is used for sending and receiving data between client and server.

Client

Picasso client has a *Server Interface*, *GUI*, *Settings*, *Tree visualization* and *2D/3D visualization* modules.

- *Server interface* module abstracts the communication to the server. Its similar to the Client Interface module in the server but is implemented from the perspective of the client.
- *GUI* module constructs the user interface including the main frame and dialog boxes and handles the user events which triggers operations on the client.
- *Settings* module stores the settings of the client persistently on disk. Settings include database settings such as address, port, schema, dbname, username, password, optimization level etc.
- *Tree Visualizer* module draws the plan trees on screen. It also supports plandiff where two plan trees are compared and differences between them are highlighted. This module makes use of an external library called JGraph[14].
- *2D/3D visualization* module draws the actual plan, plan-cost and plan-cardinality diagrams, which are either 2D or 3D color coded surfaces. This modules makes use of an external library called Visad[13].

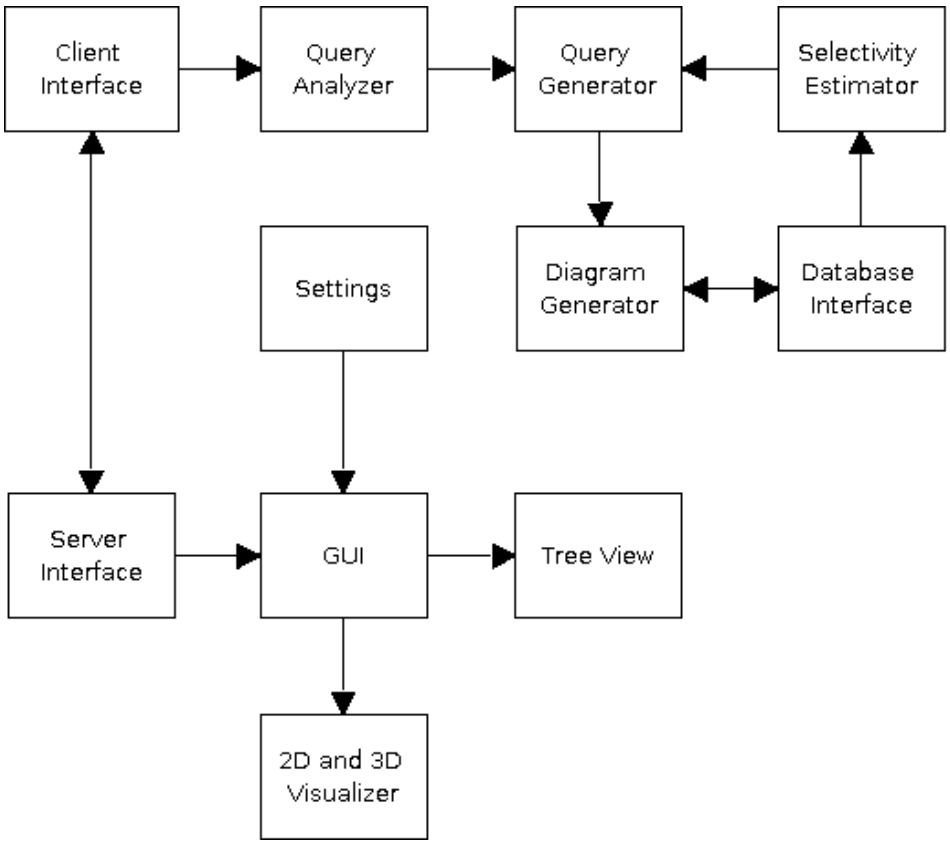


Figure 3.2: Component Diagram

Chapter 4

Picasso Server Design

```
QIDMAP(QTID, QTEXT, QTNAME, RESOLUTION, DIMENSION, EXECTYPE, SCALETYPE,
         OPTLEVEL, MACHINE, GENTIME, DURATION)
PlanTree(QTID, PLANNO, ID, PARENTID, NAME, COST, CARD, OPTIONS)
PlanTreeArgs(QTID, PLANNO, ID, ARGNAME, ARGVALUE)
PlanStore(QTID, QID, PLANNO, COST, CARD, RUNCOST, RUNCARD)
SelectivityMap(QTID, QID, DIMENSION, SID)
SelectivityLog(QTID, DIMENSION, SID, PICSEL, OPTSEL, ACTSEL, CONST)
```

Figure 4.1: Picasso Schema

Picasso server waits for clients to make connection. Once connection is established, the server sends a unique clientId to the client for all future communication between client and server. When the server receives client requests, it takes action on the message and sends back status and error messages as appropriate, and finally sends back the response message, which marks the end of service of the current request. There are different kind of requests which are discussed in detail later in the protocol section. If more than one request comes to the server concurrently and are not mutually exclusive with the current set of operations under progress, the server will operate on the request parallelly by starting a new threads. If there is a conflict due to exclusive use of the database by any of currently running thread, the server queues the request and sends back a response informing the client that request it sent has queued up. Once the server completes the current blocking operation, it will dequeue next request and start

operating on it.

When a *Picasso diagram* is requested, server will search the database to see if its already generated and stored. If already stored, server will send back the *Picasso diagram*. Otherwise the query template is passed to the *Query Analyzer* module which parses the *Query Template* and collects information on the selected predicates like the *attribute name*, *relation name* and *schema name*. Server then queries the database to get statistics on the selected attributes. For generating the *Picasso diagram*, server uses the *Query Generator* module to generate queries with appropriate constants in the selected range predicates. The selectivity values are varied either uniformly or exponentially in the selectivity space and the constants are generated for each of these selectivity values. Picasso submits the generated query to the optimizer and obtains the optimal plan according to the optimizer from database. Picasso then matches this plan with all the distinct plans that are found already. If no matching plan is found, a new plan number is assigned to the new plan and is stored in memory for future comparison. A hash value is generated for each plan which is used for the actual comparison of plans. The hash function used is provided by java standard library. This approach is much more efficient than comparing the entire tree structure. Picasso stores all generated information such as selectivity values, constants used, plans, cost, cardinality etc in the database for later retrieval.

4.1 Database Schema

The schema of Picasso related relations where we store the generated data is shown in Figure 4.1.

- *QIDMAP* stores meta information about each *Picasso diagram* such as the query template, query name, resolution, dimension, optimization level, machine name, execution type, scale type, time of generation and time taken for generating the diagram. Each entry in *QIDMAP* denotes a *Picasso Diagram*.

- *Plantree* stores the representative plan trees for each of the distinct plans in a *Picasso diagram*. Each tuple in Plantree represents a node in a single plan. It has an id as well as

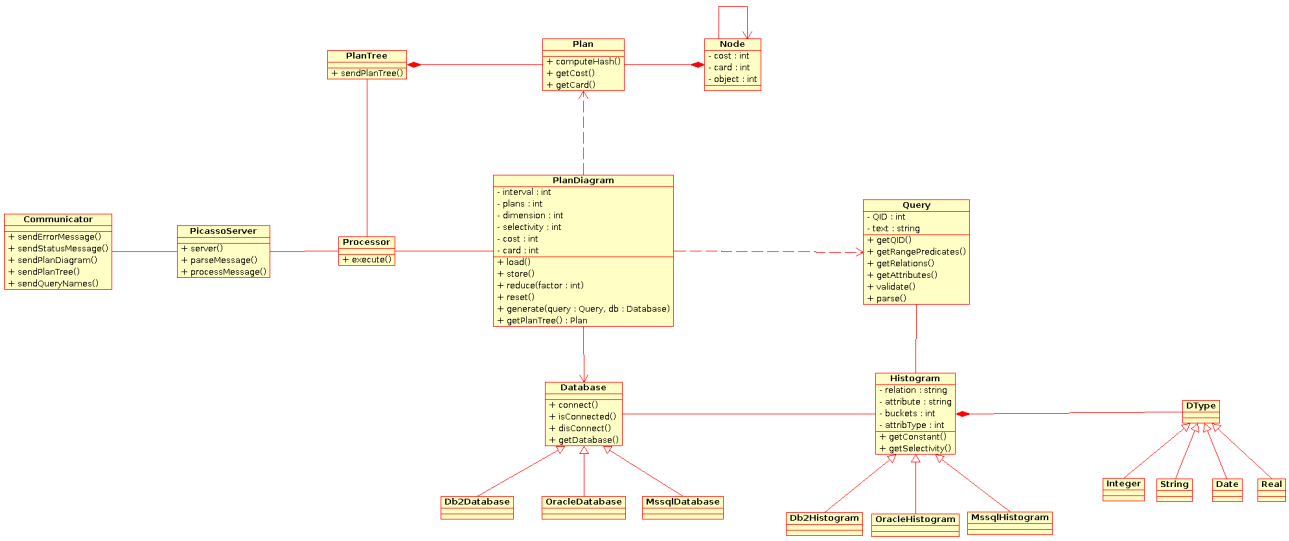


Figure 4.2: Simplified Class Diagram

a parentId which refers to the id of its parent node. This structure allows storing plan tree as a relational table.

- *PlanTreeArgs* store the sub-operator level attributes for each node in the plan if any. There can be multiple sub-operator level information for a single node, each of them resulting in a tuple in *PlanTreeArgs*.

- *PlanStore* stores the bulk of *Picasso diagram* data. This include plan number, cost and cardinality for each of the points in the selectivity space. For a *Picasso diagram* with resolution x and dimension y, number of entries in *PlanStore* is x^y.

- *SelectivityMap* maps each entry in planstore to entries in the *SelectivityLog*. Number of entries in *SelectivityMap* is same as that of *PlanStore*.

- *SelectivityLog* stores the Picasso selectivity, optimizer selectivity and actual selectivity values for the constants which are used while generating the plan diagram. For a *Picasso diagram*, number of entries in SelectivityLog is $x * y$ only.

4.2 Class Diagram

The class diagram for the server is shown in Figure 4.2. The *Communicator* class is responsible for the network communication between client and server including sending error and status messages. The *PicassoServer* class holds the top level loop of receiving requests, parsing, processing and sending the reply message. The *Processor* class starts a new thread of execution for processing each request from the *PicassoServer*. It supports stopping current operations by terminating threads whenever a 'stop operation' message is received from the client. *PlanDiagram* class is a central class which generates and retrieves different *Picasso diagrams* from the database. *PlanDiagram* creates a concrete *Database* instance to communicate with the specified database and then instantiates the *Query* class which acts as the query analyzer and generator. *PlanDiagram* generates plan number, cost and cardinality information for each point in the selectivity space. It then stores the generated data, along with unique plan trees, persistently on the same database. *Query* object uses a concrete instance of *Histogram* class, for fetching the statistics associated with the predicate attribute. This is used to generate the appropriate constants corresponding to the selectivity values of each points in the selectivity space. *Histogram* works with *Datatype* objects to be independent of the datatype of the predicate attributes.

4.2.1 Design Patterns

- *Factory Method* is used to instantiate *Datatype* objects, *Database* objects and *Histogram* objects. Database and Histogram objects are created depending on the type of database *Picasso* deals with and Datatype is determined based on the type of attribute upon which selectivity is varied.

- *Command* design pattern is used for the communication between client and server. Client constructs a command object and sends it to the server which then operates on the command. This pattern makes it easy to queue client requests if the server is busy working on a mutually exclusive operation.

- *Strategy* design pattern is used for making Database and Histogram operations independent of the specific methods of obtaining information and specific algorithms and queries

for different type of database systems. This pattern makes it easy to add support for a new database systems to *Picasso*.

4.3 Selectivity Estimator

Estimating the constant corresponding to given selectivity for a predicate is central to the operation of Picasso. This operation is infact the reverse of what databases query optimizers normally do where it estimates the selectivity from the constant. Predicate attributes can be of different datatypes which calls for abstraction of operations on data. This is explained in detail later.

There are three different notion of selectivity from the point of view of Picasso. For the sake of ease of description we will use '$\leq$' as the operator applied on the range predicate.

- *Picasso Selectivity* is the selectivity values Picasso wants to use for the attribute. Picasso makes use of histograms in the database to compute a constant corresponding to this selectivity. The underlying data distribution sometimes renders some selectivity impossible, but Picasso still attempts to find a constant that is closest to the desired selectivity.

- *Optimizer Selectivity* is the optimizers estimate of selectivity for a given constant. Optimizer uses the statistics collected in the database to do the estimation. Ideally Picasso selectivity and optimizer selectivity should be same, but we have seen slight differences in the estimates mostly triggered from data distribution and peculiarities of histograms on certain databases.

- *Actual Selectivity* is the real selectivity for the given dataset and constant for an attribute. This is deterministic and is a function of dataset only. The other two selectivity notions are talking about estimates of the *Actual Selectivity*. *Actual Selectivity* can be computed by executing the following query

```
select count(*)
from table
where attrib <= Constant
```

4.4 Datatypes

The datatypes of attributes are abstracted away using the Datatype class. The interface supports comparison and interpolation operation which are the only operations needed for *Picasso*. Any concrete implementation has to define these operators for the particular datatype being implemented. Implementation of *Integer* and *Real* datatype is trivial as it is well defined. *Date* is also implemented by representing it as a long integer which holds the number of milliseconds elapsed since the beginning of January 1st 1970.

4.4.1 String Datatype

Implementing a concrete class for the Datatype interface which supports String is not trivial. The comparison operation is defined on *Strings* based on the lexicographical ordering of characters in the alphabet. But interpolation operation is not defined properly on *Strings*. For *Integer* and *Real* if *low* and *high* are the bounds and *factor* is the interpolation factor, then the interpolated value can be found as follows

$$value = low + (high - low) * factor \tag{4.1}$$

But this will not work with strings since multiplication is not a defined operation on Strings.

Our approach tries to approximate the variation in characters of the bounding strings and use it as the base of variation in the character class. Then we define a mapping from strings to real numbers between zero and one. Using this mapping we can convert the lower and upper bound strings to the corresponding real numbers. Using this numbers we can use equation 4.1 to find the real number corresponding to target string. Then we can use the inverse function of the mapping to generate the desired string.

For example consider a table with attribute telephone number. Here there are only 10 possibilities for an individual character. For attributes that include upper case, lower case or other special characters, the variation on a single character can be high [a-Z]. In other words, if you treat string as a number with high base, the base may vary widely between different kind of attributes. Based on the above base, the interpolated value can vary.

To minimize the impact of this problem of varying base of strings, we have used the approach adopted by *PostgreSQL* relational database. The algorithm of string generation given lower bound upper bound and selectivity within the bucket is given in Figure 4.3.

4.5 Protocol

Picasso uses Java object serialization for communication between client and server. This simplifies the protocol quite a bit. Upon starting the client, it tries to connect to Picasso server and then requests a clientId. Every message communicated between client and server will have this clientId which is associated with all the client state information. Picasso makes use of request-response paradigm. Some requests takes a long time to complete and those typically sends status messages indicating progress of operation.

The list of message types are

- *GET_CLIENT_ID* is used just after the client connects to the server requesting a new clientId. The server responds by supplying an unused clientId for future communications.
- *CLOSE_CLIENT* is used when the user closes the client window. This frees up the clientId for reuse and server removes all client specific state information stored on the server.
- *READ_PLAN_DIAGRAM* is used to read plan diagram information from the server. If the plan diagram is not already generated, server responds by setting the command to TIME_TO_GENERATE and sends the expected time needed to generate the requested plan diagram.
- *GENERATE_PLAN_DIAGRAM* is used to request the server to generate a plan diagram. This is used only after first trying to read plan diagram and getting a TIME_TO_GENERATE message.
- *TIME_TO_GENERATE* is the response of server when the client requests a plan diagram which is not already generated.

- *GET_PLAN_TREE* is used to get the representative plan tree which is stored persistently in the database. The plan is identified by the plan number.
- *GET_COMPILED_PLAN_TREE* is used to get the plan tree at given selectivity. Server constructs the query corresponding to the given selectivity and generates the plan at the given point and sends it back.
- *PROCESS_QUEUED* is sent from the server if server is already using the same database for some mutually exclusive operation. Server queues the message and later operates on it as the previous requests are satisfied.
- *DELETE_PLAN_DIAGRAM* is used to delete all persistently stored data for a plan diagram.
- *GET_PLAN_DIAGRAM_NAMES* is used to query the server for getting a list of plan diagrams that are already generated and persistently stored on the database.
- *STOP_PROCESSING* is used to abort the current requested operation from the client. Server removes all partly stored data and stops the thread of execution cleanly.

```
Interpolate ( low, high, factor )
low    : String
high   : String
factor : double
  rlow = smallest character in low and high
  rhigh = biggest character in low and high
  // If range includes major char ranges,
  // make it include all
  if (rlow <= 'Z' and rhigh >= 'A')
    if (rlow > 'A')  rlow = 'A'
    if (rhigh < 'Z')  rhigh = 'Z'
  if (rlow <= 'z' and rhigh >= 'a')
    if (rlow > 'a') rlow = 'a'
    if (rhigh < 'z') rhigh = 'z'
  if (rlow <= '9' and rhigh >= '0')
    if (rlow > '0') rlow = '0'
    if (rhigh < '9') rhigh = '9'
  //Remove any common prefix of low and high
  lVal = To_Real ( low, rlow, rhigh )
  hVal = To_Real ( high, rlow, rhigh )
  newVal = lowVal + (hVal - lVal) * factor
  newStr = To_String (newVal, rlow, rhigh)
  return newStr

To_Real ( val, rangelo, rangehi )
val      : String
rangelow : character
rangehi  : character
  num = 0
  denom = 1
  base = rangehi - rangelo
  For each character ch from right end of val
    if (ch < rangelo) ch = rangelo - 1
    else if (ch > rangehi) ch = rangehi + 1
    num = num + (ch - rangelo) / denom
    denom = denom * base
  return num

To_String ( val, rangelo, rangehi )
val      : String
rangelow : character
rangehi  : character
  base = rangehi - rangelo
  newch = First character in newString
  while ( val > eps )
    newch = rangelo + floor ( val * base )
    val = val - floor ( val * base ) / base
  return newString
```

Chapter 5

Server Implementation

Picasso Server is implemented as a multi threaded Java application and uses JDBC for accessing database servers. Object serialization is used for communication with the client. The database supported by *Picasso* are *IBM DB2*[16], *Oracle*[19], *MS Sql Server*[18], *Postgres*[20] and *Sybase*[22].

5.1 Selectivity Estimator

Database optimizer estimates the selectivity of a range predicate using uniform distribution assumption if distribution statistics are not available, or by making use of distribution statistics. Distribution statistics are usually stored as *Histograms* [7] in databases. The generation, storage and representation of this information varies widely between database systems. To estimate the constant for a given selectivity Picasso needs to do the inverse of what database optimizer does.

In Picasso, the central abstract class for selectivity estimation is the *Histogram* class. There are concrete classes like *DB2Histogram, OracleHistogram* etc. which inherits from the *Histogram* class and implements the interface defined by the *Histogram* class. All other modules use the *Histogram* interface to compute the constant corresponding to a given selectivity for an attribute.

Histograms work with *Datatype* objects which enables histogram implementation to work independent of the datatype of the attribute. *Datatype* implements comparison and interpolation

logic for the specific datatypes. There are four concrete implementation in *Picasso* for the Datatype interface, which are *Integer*, *Real*, *Date* and *String*. The Histogram object holds a vector of value and frequency corresponding to the value and the selectivity and associated constants given the resolution.

DB2

DB2 stores two types of Histograms called quantile histogram and frequency histogram. Frequency histogram stores the attribute value, frequency pairs for N most frequent attribute values where N defaults to 10 and can be specified by DBA. Frequency histogram is used to estimate the selectivity of equality predicates. Quantile histogram is an equidepth range histogram which is used by the optimizer to estimate selectivity of range predicates. DB2 uses 20 buckets by default to approximate data distribution. This information is stored on the system table SYSIBM.SYSCOLDIST.

Oracle

Oracle has a single histogram which can act as either frequency histogram or equidepth histogram. Oracle uses the frequency version of histogram if the number of unique values of the attribute is not high. It switches to equidepth histogram if domain is large and number of unique values crosses a threshold. Default value of this threshold is 75 which will be the number of buckets in equidepth histogram. Oracle provides the view, all_tab_histogram, to read the histogram information.

MS Sql Server

MS Sql server has a mix of frequency and equidepth histogram. The frequency of bucket boundaries is specified along with the number of tuples in the bucket. The number of buckets can go up to 200 in Sql Server. Histograms by default are generated with sampling in MS Sql Server. The stored procedure DBCC SHOW_STATISTICS is used to extract histogram information from MS Sql Server.

Postgres

PostgreSQL maintains histograms which is a mixture of end biased and equidepth histograms. Histograms are stored in a relation named *pg_stats* catalog table. The most frequently occurring values are stored as an array in the *most_common_vals* column. The equi-depth histogram is stored in the form of two arrays which store the frequency of corresponding buckets and the bounds of the buckets respectively. These are named as *most_common_freqs* and *histogram_bounds*. The default number of buckets is 10 on Postgres.

Sybase

Sybase also stores statistics on relational tables such as *systabstats* and *sysstatistics*. But unfortunately the fields of these relations are cryptic and the values present in it is understood only by the database itself. Therefore an external stored procedure, *hist_values*, available from a Sybase forum [23] is used to get the distribution statistics.

```
select  source_id, target_id, operator_type, object_name,
  explain_operator.total_cost, stream_count, argument_type,
  argument_value
from explain_statement, explain_stream
  LEFT OUTER JOIN explain_operator
  ON  (source_id = explain_operator.operator_id
    or (source_id=-1 and source_id = explain_operator.operator_id))
    and explain_operator.explain_time = explain_stream.explain_time
  LEFT OUTER JOIN explain_argument
  ON  explain_operator.operator_id = explain_argument.operator_id
    and explain_stream.explain_time = explain_argument.explain_time
where explain_stream.explain_time = explain_statement.explain_time
  and explain_statement.explain_level='P' and queryno=QNO
order by target_id, source_id, argument_type asc, argument_value asc
```

Figure 5.1: DB2 plan query

5.2 Plan Representation

A plan is stored as a vector of nodes which consists of id, parentid, name, options, cost and cardinality. Tree can be captured by having parent id of a node pointing to the id of its parent. Plan class computes a hash value for a plan based on the following equation.

$$\Sigma id_i * pid_i * (hash(nodename_i) + hash(nodeattr_i)) \tag{5.1}$$

Plans are compared based on this hash value which will be different if there is any structural as well as node name or options.

5.3 Plan Retrieval

Retrieving the plan which optimizer chooses from the database for a given query is one of the central operations done by *Picasso Server*. Different database has different ways of supplying this information. The approaches we used for DB2, Oracle and MS Sql Server, Postgres and Sybase are described below.

DB2

DB2 has seven explain tables which stores the plan information. We can supply a query preceded with the clause 'explain plan for' for storing the plan information in these tables. The earlier implementation of retrieving plan from DB2 read these tables one by one for fetching the information. This was too slow compared to the methods provided by other databases. We were able to speed up the retrieval by using a complex SQL query given in Figure 5.1.

Oracle

Oracle has a single plan table which stores the plan tree information. Like DB2, Oracle supports populating this table with the plan tree using 'explain plan for' clause before query. The schema for Oracle PLAN_TABLE is PLAN_TREE(id, parent_id, operation, object_name, cost, cardinality, options).

MS Sql Server

MS Sql Server doesn't store the plan on relational tables. It doesn't even provide a view for accessing this information. The approach MS Sql Server has taken is to first execute the command 'set showplan_all on' before executing the query. Then instead of the result sets, queries are answered with the plan as *ResultSet* which can be used by Picasso to populate the Picasso datastructure for plan representation.

Postgres

PostgreSQL provides the plan in textual form which is returned when the explain clause is used in front of the query. The format has nested statements to represent the plan structure and associated costs and cardinalities. Picasso needs to parse these information to populate its plan datastructure.

Sybase

Sybase provides the plan in textual and XML form which needs to be parsed by Picasso. Since parsing XML is easier, we chose to use the XML output of Sybase which can be obtained by the following set of commands.

```
dbcc traceon(3604)
set plan for show_best_plan_xml to client on
set noexec on
```

5.4 Query Generation

Query generator gets a concrete histogram object for the given database and use it to compute constants for the selected attributes in the query. The granularity is defined by the resolution, which determines how many unique constants are generated for a single dimension. Query generator generates queries by modifying query template with the generated constants. These

queries are fired to the database for optimizing and the resultant plan is obtained by the plan diagram generator.

5.5 Picasso Parser

Picasso needs to parse the query template to determine the selected predicates which are marked by ':num' format. It also needs to identify the relation and schema associated with the attribute in the predicate along with the operation. To achieve this a regular expression based parser is implemented which collects the list of relations in the query and attribute names in the selected predicate in a tree structure. Tree structure is needed to handle nested queries correctly. The attribute name is searched in all relations present in the current scope to find the associated relation and schema. Another complication the parser needs to handle is aliases for relation names, which can be used to explicitly specify the association of an attribute to a relation.

5.6 Execution picasso diagram

Execution picasso diagram is similar to plan-cost and plan-cardinality diagrams, but instead of using optimizers estimate values for cost and cardinality, the time taken and cardinality of result set after executing the query is used. Generating execution plan diagram is a very costly operation as we have to actually execute the query. The time taken to execute the query, which is considered as the cost, is system dependent and may vary based on external factors such as load on the system etc. Execution plan diagram is very useful for comparing the effectiveness of the cost model of optimizer in real world.

5.7 Progress and Time Estimator

Generating plan diagrams and execution diagram are costly operations. Therefore giving feed back of estimated time to the user is a highly desirable feature. In Picasso we do static and dynamic estimates of time to complete. Static estimator is based on firing a few sample queries at the corners of the N-dimensional hypercube in selectivity space and taking the average value

to compute the estimate of explaining a single query. The product of this value and number of total points in the hypercube gives an estimate of time to generate the plan diagram.

Dynamic estimate is based on the number of queries fired so far and total queries. Picasso also tries to take into account the difference in time for explaining the query at different corners based on the static estimate.

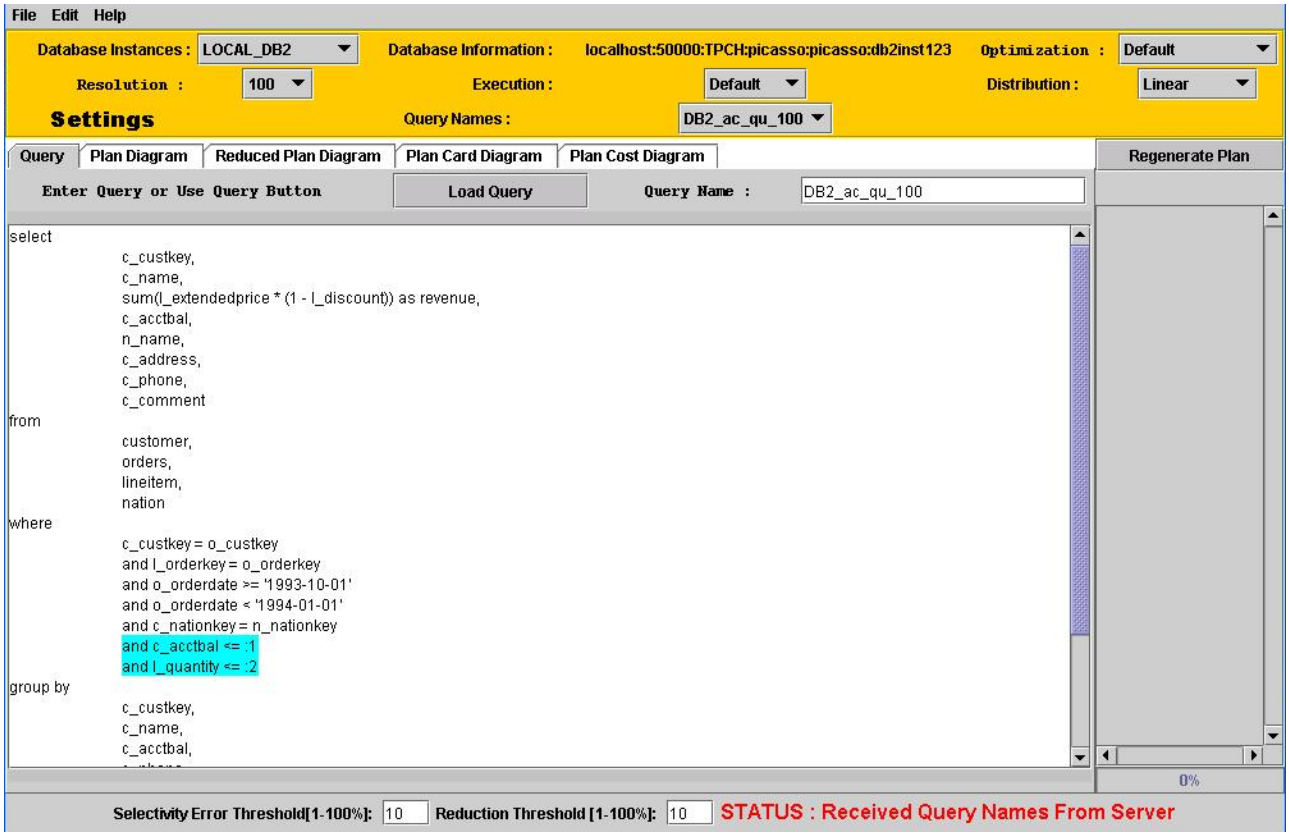


Figure 5.2: Client Screen Shot

5.8 N dimension support

Picasso fully supports generating N-dimensional plan diagrams if the query has N range predicates. This is achieved by using an index array. Picasso lays out the plan number, cost and cardinality information to single dimension for storage and communication. The index array is used to calculate the position of each point in the single dimension layout. Support for N dimension has made the implementation of generation, retrieval, storage of *Picasso diagrams*

more complex. This feature has effects on the database schema and time estimator since all these have to be generic in terms of the number of dimensions assumed.

5.9 Summarizing

Picasso supports generating plan diagrams of different resolutions from higher resolution versions if a higher resolution version is already generated. This allowes us to find out the rate at which number of unique plans grow as we increase the resolution without explicitly generating plan diagrams at various resolutions.

5.10 Slicing

N dimensional plan diagrams cannot be visualized by the client. Therefore the model we chose was to slice the hypercube and send only a two dimensional slice of the hypercube. This needs the selectivity values for $N-2$ dimensions and the unspecified dimensions are varied to obtain the slice.

5.11 Selectivity Distribution

Picasso supports both uniform and exponential distribution of selectivity values for data points in the selectivity space. This is because often more plans are present in the low selectivity regions and user may like to explore that part of selectivity space at higher granularity. The exponential distribution of selectivity values places more points in the low selectivity region for finer granularity in that area and becomes coarser at high selectivity region where fine granularity may not be required.

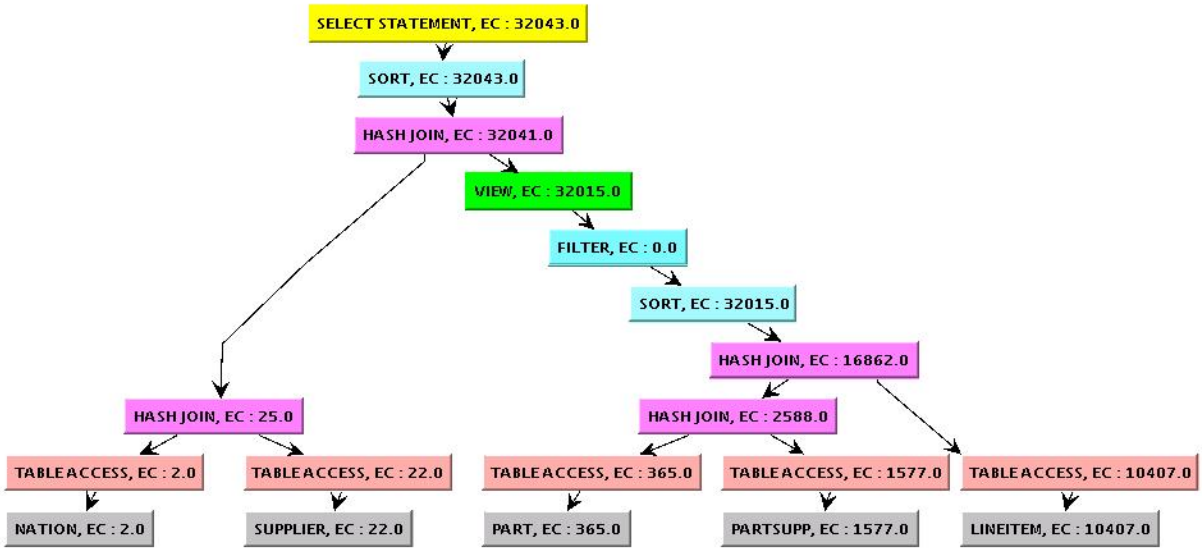


Figure 5.3: Plan Tree Screen Shot

Chapter 6

Picasso Client

Picasso client is implemented in Java. 2D/3D visualization is done using the library Visad. Plan trees are drawn using another library JGraph. A screenshot of the interface is provided in Figure 5.2. The top panel allowes to choose the database instance that is used, optimization level, resolution, execution type and distribution. The query panel is used to input, load and show SQL query. An SQL query along with the above settings is identified with a query name which can be set in a text box. When any of the tabbed panes such as plan panel, plan-cost panel, plan-cardinality panel or reduced plan panel is accessed, a request is send to the server for reading the plan diagram corresponding to the query name. If its not already generated, server responds with the estimated time to generate the diagram. If the client accepts diagram is generated and shown. A legend panel on the right side shows the color codes for different plans in these diagrams. Right clicking on the plan diagram will show the plan tree. A screen shot of plan tree is shown in Figure 5.3.

Chapter 7

Conclusions and Future work

The Picasso server and client is implemented which serves as an excellent tool for characterizing and analyzing relational query optimizers. The client server architecture allows running the server on high end machines while the client runs on low end desktops.

Like any software, Picasso can be improved by adding several database management functionality such as creating explain plan tables or creating statistics directly from the Picasso tool. Currently N dimensional diagrams are generated fully before a slice of it can be visualized. A nice feature in this regard will be to support generating only the requested slices of N Dimensional diagram. Feeding plans generated by one database to another and compare the cost and cardinality estimate for the same set of plans over the selectivity space allows more meaningful comaparisons across databases.

Bibliography

[1] R. Cole and G. Graefe, “Optimization of dynamic query evaluation plans”, *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1994.

[2] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price, “Access Path Selection in a Relational Database Management System”, *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1979.

[3] M. Jarke and J. Koch. Query optimization in database systems. ACM Computing Surveys, 16(2):111-152, June 1984.

[4] A. Ghosh, J. Parikh, V. Sengar and J. Haritsa, “Plan Selection based on Query Clustering”, *Proc. of 28th Intl. Conf. on Very Large Data Bases*, August 2002.

[5] N. Reddy and J. Haritsa, “Analyzing Plan Diagrams of Database Query Optimizers”, *Proc. of 31st Intl. Conf. on Very Large Data Bases*, September 2005.

[6] N. Reddy, “Next Generation Relational Query Optimizers”, *Master’s Thesis, Dept. of Computer Science and Automation, IISc Bangalore*, June 2005.

[7] Y. Ioannidis, V. Poosala. *“Histogram-Based Solutions to Diverse Database Estimation Problems”*. Data Engineering Bulletin, 1995.

[8] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *”Design Patterns, Elements of Reusable Object-Oriented Software”*

[9] Grady Booch, James Rumbaugh and Ivar Jacobson *”The Unified Modeling Language, User Guide”*

[10] *http://dsl.serc.iisc.ernet.in/*

[11] *http://dsl.serc.iisc.ernet.in/projects/PICASSO*

[12] *http://dsl.serc.iisc.ernet.in/projects/PLASTIC*

[13] *http://www.ssec.wisc.edu/ billh/visad.html*

[14] *http://www.jgraph.com/*

[15] *http://publib.boulder.ibm.com/infocenter/db2luw/ v8//index.jsp*

[16] *http://www-306.ibm.com/software/data/db2/udb/v8/*

[17] *http://ibm.com*

[18] *http://www.microsoft.com/sql/techinfo/productdoc/ 2000/books.asp*

[19] *http://www.oracle.com/technology/products/ oracle9i/index.html*

[20] *http://www.postgresql.org*

[21] *http://www.ibm.com/software/data/informix/*

[22] *http://www.sybase.com/products/ informationmanagement/adaptiveserverenterprise*

[23] *http://www.sybase.com/support/newsgroups*

[24] *http://www.tpc.org/tpch*