PLASTIC: Reducing Query Optimization Overheads through Plan Recyclying

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of

Master of Engineering

in

Faculty of Engineering

by

Vibhuti Singh Sengar

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

MARCH 2003

Contents

List of Figures

Chapter 1

Abstract

We present PLASTIC, a tool intended for use by the query optimizers to significantly amortize the optimization cost. The tool groups similar queries into clusters and uses the optimizer-generated plan for the cluster representative to execute all future queries assigned to the cluster. In this report, we present architecture, improvements to feature vector and distance function in PLASTIC, and implementation on commercial database systems DB2 7.0 and Oracle 9i. The main features of our prototype design are the following: First, it is non-intrusive, not requiring any modifications to the query optimizer itself. Second, it is completely contained within the database system, without any dependency on the underlying operating system. Third, it is modular, supporting localized upgrades to the tool components. Fourth, the interface to the host database system is restricted to a few modules, facilitating portability to other optimizers and platforms. Fifth, visual interfaces for auto generating, viewing, and reorganizing query clusters are provided to assist the database administrator in tuning the system. Finally, an automated mechanism is provided for creating and visualizing "plan diagrams" and "cost diagrams".

Chapter 2

Introduction

SQL, the standard database query language is a declarative language, and it does not specify how a SQL query will be executed [10]. There may be many different ways to execute a query but some of them may be less inefficient than others. In RDBMS, the

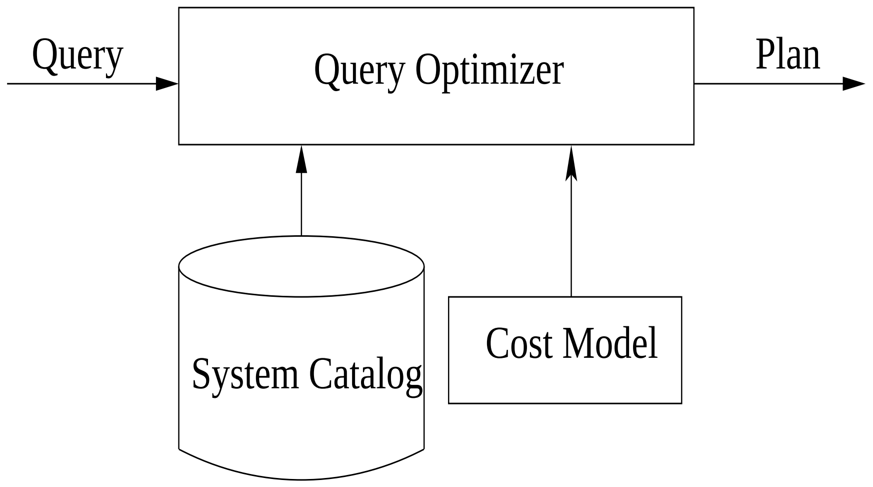


Figure 2.1: The Current Optimizer Architecture

query optimizer component decides the query plan, showing how query will be executed. A query submitted to system is passed to the query optimizer. Typical architecture of the query optimizer is given in Figure 2.1. The query optimizer consults system catalogs

[1] and based on cost model [2] decides the query plan, which is used to execute the query. Query optimization is well-known to be a computationally intensive process since a combinatorially large set of alternative plans have to be considered and evaluated in order to find an efficient access plan for query [9]. This is especially so for the complex queries that are typical in current data warehousing and mining applications, as exemplified by the TPC-H decision support benchmark [5]. The inherent cost of query optimization is compounded by the fact that typically each new query that is submitted to the database system is optimized afresh. PLASTIC (PLAn Selection Through Incremental Clustering), is a value-addition tool for the query optimizers that *amortizes* the cost of query optimization through the *reuse* of plans generated for earlier queries [2]. More specifically, the tool stores a cache of plans[3] and tries to assign one of these pre-existing plans to the new in-coming query with the expectation that the selected plan would be the same as that generated by the query optimizer. If no suitable assignment can be made, then optimization process actually carried out and the abstract operational representation of newly-generated plan is added to the cache for future use.

PLASTIC groups similar queries into clusters and uses the optimizer-generated plan for the cluster representative to execute all future queries assigned to the cluster. Query similarity is evaluated based on a comparison of query structures and the associated table schemas and statistics, and a classifier is employed for efficient cluster assignments. Experiments with a variety of queries on commercial optimizers shows that PLASTIC predicts the correct plan choice in most cases, thereby providing significantly improved query optimization times. Further, even when errors were made, the additional execution cost incurred due to the sub-optimal plan choices was marginal.

Apart from the obvious advantage of speeding up optimization time, PLASTIC also improves query execution efficiency since it makes it possible for optimizers to always

[1]System catalog stores information about all the data in database ie. size of tables, histograms, and indexes.

[2]Refers to functions needed to evaluate cost associated with different possible operations which can be used to a query.

[3]Here we refer to plans as *plan templates* explained later.

run at their highest optimization level[4] as the cost of such optimization is amortized over all future queries that reuse these plans. Yet another important advantage is that the benefits of “plan hints”, a common technique for influencing optimizer plan choices for specific queries, automatically percolate to the entire set of queries that are associated with this plan. Lastly, since the feature vector includes components based on database statistics the association of queries with clusters is dynamic, so the plan choice for a given query is *adaptive* to the current state of the database.

The prototype design of the PLASTIC implementation is fully contained in the database system, non-intrusive with respect to the optimizer, easily portable across platforms and optimizers, extendible to incorporate changes to the tool modules, and with visual interfaces for easily understanding and analyzing the tool’s performance.

[4]Some optimizers like DB2 provide optimization level which gives measure of time spend in query optimization.

Chapter 3

Overview of PLASTIC

The inherent cost of query optimization is compounded by the fact that typically each new query that is submitted to the database system is optimized afresh. While current commercial query optimizers do provide plan caching (for example, the Oracle 9i optimizer provides "stored outlines" as a mechanism for preserving queries and execution plans [6]), the query matching is extremely restrictive – only if the incoming query has essentially an *identical match* with one of the stored queries is the associated plan used.

The goal of PLASTIC is to support plan reuse for *similar* queries, not just identical queries, thereby realizing a much higher degree of plan reuse. In fact, PLASTIC can identify potential plan similarities between queries in spite of *differences in projection, selection and join predicates, as also in the query tables themselves.* Therefore, it promises to significantly improve the utility of plan caching.

To achieve this functionality, the following approach is taken: First, a query *feature vector* is defined comprised of information that can be determined from the query and from the catalogs of the RDBMS. The feature vector is described in section 3.1.

Next, a similarity function which takes a pair of query feature vectors as input and quantitatively computes their separation in feature space is defined. A threshold value for this separation is used to decide whether or not the two queries are similar.

Then, using this similarity definition described in section 3.2, *query clusters* are dynamically formed in an incremental manner, with the distance threshold determining the maximum stretch of the cluster. Each cluster has a *representative* for whom the execution plan, as determined by the optimizer, is persistently stored. This plan is used to execute all future queries that are assigned to the cluster. Finally, when a sufficient number of clusters have been formed, a classifier is constructed on the clusters to support efficient identification of the cluster to which a new query may belong, thereby also determining its execution plan.

An important detail that we have glossed over in the above discussion is that it is the *plan templates*, and not the plans themselves, of the cluster representatives that are stored persistently. A plan template is a query execution plan wherein all the database operators (e.g. TABLE SCAN, SORT, MERGE-JOIN, RANGE SCAN) are retained but the specific values of inputs to these operators such as table names, index names, attribute names, etc. have been replaced by variables. When a plan template is assigned to a new query, these variables are instantiated with the values appropriate to the query.

3.1 Feature Vector

Note that we are constrained to use only features that can be extracted directly from the inputs to the query optimizer, namely the query and the metadata from the system catalogs, but not any intermediate computation since otherwise we might wind up repeating the optimization exercise.

The specific features we choose are divided into two classes: *Structural*, which are determined from the the query and associated schema-related meta-data, and *Statistical*, which are determined from the table statistics available in the system catalogs. These features, which are described in the remainder of this section, were arrived at after an extensive study of the characteristics of the plans generated by the commercial optimizers over a broad spectrum of queries.

3.1.1 Structural Features

We will use Figure 3.1 to motivate and explain some of the structural features. This figure shows two graphs which represent two different queries on six tables each. In these graphs, the nodes A, B, $\ldots$, F and P, Q, $\ldots$, U represent the tables and the lines between them represent the *join* predicates relating them. (The distinction between the full and dashed line types is explained later in this section.) The structural features are the following:

Degree of a Table (DT): This is a vector containing relation numbers to which particular relation is joining. For storage efficiency we store it as a number. This feature is included since it plays a role in positioning the table within the join tree in the access plan of the corresponding query.

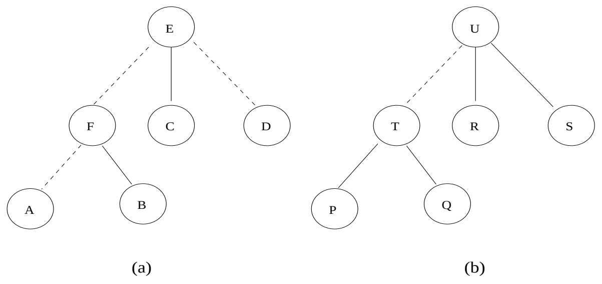


Figure 3.1: Degree Graphs

Degree-Sequence of a Query (DS): This is a (derived) compound feature that is based on the DT feature – specifically, it is a *non-increasing vector* composed of the DTs of all the tables involved in the query. For example, the DS for both the queries shown in Figure 3.1 is (3, 2, 1, 1, 1, 1).

Join Predicate Index Counts (JIC): A join predicate is said to have an *index characteristic* of 0, 1 or 2, depending on whether there are 0, 1 or 2 indexed attributes, respectively, in the join predicate. For each query, a count of the number of join predicates, with respect to each characteristic value, is evaluated since these counts help to determine whether an index can be used for the join.

Predicate Counts of a Table (PC): A predicate can be, as per the definition in the

System R optimizer [11], either *SARGable* or *Non-SARGable*, the primary difference being that the former can be evaluated through indexes, whereas the latter is incapable of using these access structures. For example, $x = 10$ is an SARGable predicate while $x <> 10$ is not.

For each table involved in the query and for each predicate type, we maintain the count of the number of such predicates operating on the table. The reason for including this feature is that an index on a table can be used only if the associated predicate is SARGable. In Figure 3.1, the dashed lines represent SARGable predicates while the solid ones represent non-SARGAble predicates. Therefore, the type counts for table E are (2, 1), implying that there are two SARGable predicate and one Non-SARGable predicate that will be evaluated on this table.

Index Flag of a Table (IF): An Index Flag is associated with every table and is set if *all* the selection predicates and projections on that table can be evaluated through a single common index. In this situation the optimizer can construct a plan that reads only the index and not the table itself.

3.1.2 Statistical Features

We now move on to the features that are based on the statistics available in the system catalogs:

Table Size (TS): It is a measure of the total size of a table and is computed as the product of the cardinality of the table and the average length of the tuples present in the table.

Effective Table Size (ETS): This is size of table required for query processing.

Putting all of the above together, our complete query feature vector definition is as shown in Table 3.1. For ease of understanding, we have separated the features into *Global Features*, which are query-wide values, and *Table Features*, which are relevant to individual tables.

Feature	Description
	Global Features
NT	Number of tables participating in the query
DS	Degree sequence of query
NJP	Total number of join predicates
$JIC[0..2]$	Number of Join Predicates with index characteristics of 0, 1 and 2, respectively
$PCsarg$	Number of SARGable predicates
$PCnsarg$	Number of non-SARGable predicates
	Table Features
DT_i	Degree of table T_i
IF_i	Boolean indicating index-only access to T_i
$PCsarg_i$	Number of SARGable predicates on table T_i
$PCnsarg_i$	Number of non-SARGable predicates on T_i
$JIC_i[0..2]$	Number of Join Predicates of index characteristic 0, 1 and 2 involving T_i
TS_i	Size of T_i
ETS_i	(estimated) Effective size of T_i

Table 3.1: Query Feature Vector

3.2 Establishing Similarity

Given our goal of clustering queries in such a way that the access plan for all queries in the cluster is the same, a straightforward answer to this query would be “Two queries are similar if the optimizer generates the same plan template for both of them”. However, this is not a practically useful definition because, as mentioned earlier, optimizers map several different kinds of queries to the same plan template, resulting in extremely heterogeneous clusters that cannot be easily characterized. For example, consider the following two queries on the TPC-H table PART:

select * from part

and

select p_brand, p_name, p_mfgr

from

part

where

p_size = 4

and p_brand = 'Brand15'

The DB2 optimizer generates the *same* plan template for both these queries although a visual inspection shows them to be quite different in both syntax and semantics. The reason for choosing the same plan is that the amount of data that is required to be processed is estimated to be similar in the two cases.

To avoid the above problem, we take a different approach in PLASTIC. That is, we try to establish a notion of similarity that facilitates both (a) efficient classification of new queries, and (b) that the plan chosen by the optimizer for the queries within a cluster is the same in the majority of the cases. Our approach, hereafter referred to as the SIMCHECK algorithm, is described in detail below.

3.2.1 The SIMCHECK Algorithm

The SIMCHECK algorithm, whose pseudocode is shown in Figure 7.6, takes as input two query feature vectors and outputs a boolean value indicating whether or not they are similar. The algorithm operates in two phases, "Feature Vector Comparisons" and "Mapping Tables". In the first phase, the feature vectors are compared for equality on the number of tables, the sum of the table degrees, and the sum of the join index and predicate counts. Only if there is equality on all these structural features is the second phase invoked, otherwise the queries are deemed to be dis-similar. The equality check is done first in order to identify dis-similar queries as early and as simply as possible. For example, it is obvious that if the number of tables in the two queries do not match, then their plans will also necessarily have to be different. Such structural feature checks are used as an effective mechanism for stopping unproductive matching at an early stage.

In the Mapping Tables phase, we attempt to establish the closest possible one-to-one correspondence between the tables of the two queries. The tables are mapped to each other in order to check whether it is possible for the optimizer to use similar plans for accessing the mapped tables. The first step in this process is to determine the sets

```
SIMCHECK ($Q1, Q2$)
   // Check that Queries have same number of Tables
1. IF NT(Q1) != NT(Q2) RETURN (Not Similar);
   // Match Query Level Semantics
2. IF DS(Q1) = DS(Q2) AND
     NJP(Q1) = NJP(Q2) AND
     PCsarg(Q1)+PCnsarg(Q1) = PCsarg(Q2)+PCnsarg(Q2)
        GO TO Line 4 ;
3. RETURN (Not Similar);

   //—— Find the Best Mapping between Tables ——
4. FOR every group $g$ of tables with the same degree
        $R_1 = T_1^1, T_1^2, ...T_1^k$  $R_1 \subseteq Q_1$
        $R_2 = T_2^1, T_2^2, ...T_2^k$  $R_2 \subseteq Q_2$
        find the mapping of compatible tables
        between $R_1$ and $R_2$ that has the minimum
        aggregate distance, $mindist_g$, with respect to
        the pairwise table distance function
        $dist_{ij}(T_1^i, T_2^j) = \frac{w_1 * |TS_1^i - TS_2^j| + w_2 * |ETS_1^i - ETS_2^j|}{max(TS_1^i, TS_2^j)}$
   //—— Compute Distance between Queries ——
5.  $TotalDist = \sum_{g \in G} mindist_g$
6.  IF $TotalDist > Threshold$ RETURN (Not Similar);
7. RETURN (Similar);
```

Figure 3.2: **The SIMCHECK Algorithm**

of *compatible* tables. For every possible pair of compatible tables, SIMCHECK checks whether their original and (estimated) effective sizes are comparable through the use of a distance function. If the outcome of the distance computations is less than a threshold value which is an algorithmic parameter, the queries are said to be similar. The notion of compatibility and the distance function are elucidated below.

Table Compatibility

We define two tables to be compatible if the degrees, join index counts and predicate counts are the same for both tables. The rationale for this notion of compatibility is explained below.

Let us first consider predicate counts. The predicate count for table E in Figure 3.1(a) is (2, 1) since there are two SARGable predicates and one non-SARGable predicate.

Similarly, for table U in Figure 3.1(b), the predicate count is (1, 2), and by our definition the tables are not compatible. This makes intuitive sense when viewed in light of the fact that if a predicate on a table is not SARGable, an optimizer cannot use an index to access that table. Thus, plans can change considerably even if the two queries differ on only a single table with respect to this criteria.

A similar and stronger argument holds for join index counts. If indexes are available for a join predicate in one query and not in the other, it is very likely that the plans for the two queries will not match. This is because if both the attributes in a join predicate are indexed and the selectivities of the tables are high then it is possible to choose a plan involving an index join. Similarly, if one of the attributes is indexed then the optimizer may choose to index on one table and fetch (table scan) on the other.

Note that even if the join index counts and predicate counts for two queries match, the plans chosen by the optimizer may differ as there are other statistical factors such as the table sizes that affect plan choices. These factors are captured in the distance function discussed next.

The Query Distance Function

After compatible tables are identified, SIMCHECK tries to establish valid one-to-one mappings between the sets of compatible tables. These mappings are then compared using their original and estimated effective sizes, through a distance function $dist_{ij}(T_1^i, T_2^j)$, where T_1^i and T_2^j are the tables whose distance is to be computed, with T_1^i denoting the i^{th} table of the first query which is to be mapped with T_2^j, the j^{th} table of the second query. The larger the distance, the lesser the similarity. In terms of the statistical features described in Section 3.1.2, the distance function is given as:

$$dist_{ij}(T_1^i, T_2^j) = \frac{w_1 * |TS_1^i - TS_2^j| + w_2 * |ETS_1^i - ETS_2^j|}{max(TS_1^i, TS_2^j)}$$

Here, w_1 and w_2 are weighting factors (with $w_1 + w_2 = 1$) that serve to calibrate the importance of the associated terms in the above equation.

Chapter 4

Architecture and Design of PLASTIC

In this chapter we are describing architecture and design goals of implementation of PLASTIC system.

4.1 The Architecture

A block-level diagram of the PLASTIC architecture is shown in Figure 4.1. In this picture, the solid lines show the sequence of operations in the situation where a matching cluster is found, while the dashed lines represent the converse situation where no match is available.

The query submitted to the system is first passed to the *Feature Vector Extractor* which also accesses the system catalogs and obtains the information required to produce the feature vector. The *SimilarityCheck* module takes this feature vector and establishes whether it has a sufficiently close match with any of the cluster representatives stored in the *Query Cluster Database.* To hasten the process of cluster identification, the module may construct a classifier, such as a decision tree or a Bayesian network, on the clusters in the database.

If a match is found, the plan template for the matching cluster representative is accessed from the *Plan Template Database.* As mentioned earlier, a plan template has database operators but does not have the specific values of the inputs to these operators. These missing values are filled in by the *Plan Generator* module based on mapping string given by *SimilarityCheck* module and the specifics of the input query.

On the other hand, if no matching cluster is found (dashed lines in Figure 4.1), then the Query Optimizer is invoked in the traditional way and the plan it generates is used for executing the query. This plan is also passed to the *Plan Template Generator* which converts the plan into its abstract operational representation and stores it in the *Plan Template Database.* For efficiency reasons, the plans may be stored in the form of signatures. Concurrently, the feature vector of the query is stored in the *Query Cluster Database.*

Periodically, the cluster database may be reorganized to suit constraints such as a memory budget or a ceiling on the the number of clusters. For example, it may be decided to purge the feature vectors and plan templates of "outlier" queries that rarely result in matches with the current query workload.

4.2 The Design

Based on the above architecture, we have developed a prototype implementation of PLASTIC. The highlights of our software organization are the following:

Non-intrusiveness: The query optimizer is treated as a black box and no internal modifications to the query optimizer code are required. Further, it does not require any special inputs beyond those already available to the query optimizer.

Database-containment: All intermediate and supporting data are stored in the database system itself without requiring any support from the operating system. For example, the Query Cluster Database and the Plan Template Database are implemented as tables in the host database system. This feature allows the tool to be directly ported to any platform on which the database system is available.

Tool Modularity: The implementation follows the modular structure of the architecture and, to the extent possible, permits localized upgrades to the modules without

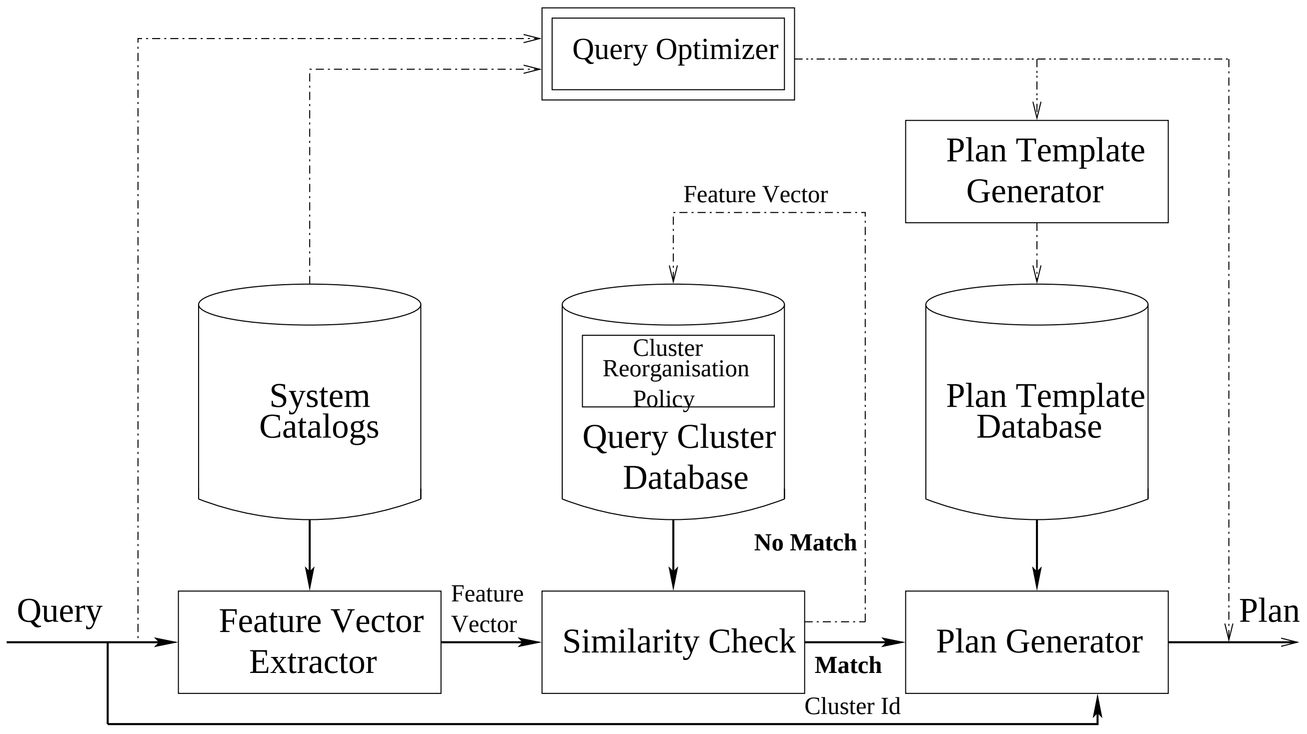


Figure 4.1: The PLASTIC Architecture

having side-effects on the other components of the tool. For example, a change in the query distance function which is part of the Similarity Checker module does not affect any of the other components of the tool. Currently, the only module which does incur major side-effects are the Feature Vector Extractor,and Query Generator. Changes to feature vector will require modifications only to feature vector classes.

Optimizer Portability: The dependencies on the specific optimizer with which integration is being achieved is localized to a few modules. Specifically, the Plan Template Generator, the Query Generator Generator, and the Feature Vector Extractor have to be customized based on the manner in which the host database system outputs plans and stores information in the system catalogs, but the remaining modules can be used directly without alteration. All the database schemas used to store clusters are independent of optimizer.

Platform Portability: The entire code has been implemented in Java, making it easily portable across different operating systems and hardware installations.

Visual Operability: The tool can be operated completely in the visual domain by the Database Administrator. For example, there are visual interfaces for both generating and viewing query cluster formations, as well as their reorganization.

Chapter 5

Value-Addition Modules

In this section we describe some additional modules that we have implemented in PLASTIC. Though not fundamental to the system these modules enhance the functioning of PLASTIC by providing interfaces to generate, reorganize, show clusters and generating ”plan diagrams”, and ”cost diagrams”.

Plan and Cost Diagram Generator: This module can be used to generate the“plan diagrams” and ”cost diagrams” for a “query template”. A query template represents a query in which some or all of the constants [1] have been replaced by bind variables. For example, each of the queries Q1 through Q22 of the TPC-H benchmark can be considered as a query template [5]. The plan diagram for a query template is the enumeration of the plans chosen by the optimizer over all points in the associated query template space. The cost diagram for a query template is the enumeration of optimizer estimated cost of the plans chosen by the optimizer over all points in the associated query template The number of dimensions of the plan diagram and the cost diagram is equal to the number of tables of a query template that have selection predicates on them. Currently, we support two-dimensional plan diagrams and cost diagrams. The DBA can specify accuracy levels depending on his requirement. At high accuracy levels, the module fires more explain queries[2] and populates the plan

[1]Here we refer to constants as values being compared in the conditions of the where clause.
[2]Explain query is used to generate optimized plan.

diagram and cost diagram more accurately. The DBA can also specify a portion of the query template space, so that the module generates the plan diagram and the cost diagram only for that portion more accurately.

This module generates the queries of different selectivities for a query template and fires explain queries for them to the query optimizer. Then it takes the optimized plans for the fired queries from the query optimizer and compares them using graph matching algorithm, based on that draws the plan diagram.

The plan diagrams and the cost diagram help the database administrator to assess the "volatility" of the plan space and thereby fine-tune the similarity function parameters. The plan diagrams and the cost diagrams can also be used by database designers to study the behavior of the query optimizers and improve the query optimizers.

Cluster Generator and Reorganizer: For a given query template, we have automated the process of "seeding" the cluster database by generating an initial set of queries through an *Auto Cluster Generator*. This module consults the data dictionary and automatically generates queries of different selectivities so as to cover the space of queries represented by the query template. To handle the case where the database administrator decides to change the similarity function threshold, resulting in new clusters, there is a *Cluster Reorganizer* that automatically creates the new clusters. If the threshold is being increased then the *Cluster Reorganizer* eliminates redundant clusters. If the threshold is being decreased then it creates new clusters so as to cover whole query template space.

Chapter 6

Implementation

We have implemented PLASTIC for two commercial RDBMS namely DB2 (Ver7.0) and Oracle (Ver 9i) in java (Ver 1.4). The code length is around 14k lines. Currently, only SPJ[1] queries are handled, but we plan to add nested and aggregate queries in future. Salient details of our current implementation are given below:
For each module there is a generic class which implements the query optimizer independent functions and gives declaration of the query optimizer dependent functions. The query optimizer specific classes for a module inherits from the generic class of that module and implements the optimizer dependent functions declared in the generic class. Due to this design, porting of PLASTIC to any other optimizer is very easy. It requires implementing optimizer specific classes inheriting from the generic classes.

Class diagram of current implementation is shown in Figure 6.1, which shows important classes in the implementation and the dependencies between them. To make the class diagram easily understandable we have not given every detail about the classes.

For example we describe, classes which implement functions of the *Feature Vector Extractor*. The *statCollect* is a generic class used to get the feature vector of a query. First, we describe the members attributes of this class. The member attribute *table-Feature* is used to store the per table feature vector [2] of each table in the query. The

[1]Here SPJ refers to select project join queries.
[2]Refers to feature vector relevant to individual table.

member attribute *globalFeature* is used to store the global feature vector [3] of the query. Second, we describe methods and processing of the *statCollect* class. Member functions *setXXX()* are called to compute the feature vector of the query. The functions *setXXX()* consult RDBMS catalogs and compute the corresponding feature of the feature vector. For example, the function *setIF()* computes the *Index Flag* of relations. The RDBMS independent *setXXX()* functions are implemented in the generic class *statCollect* and optimizer dependent functions are implemented in the optimizer dependent classes like *statCollectOracle*.

We are only describing two functions *setETS()* and *setTS()*. The member function *setETS()* computes *ETS* feature of the per table feature vector. The *ETS* is computed as product of *project ratio* [4] and overall selectivity of the relation. Overall selectivity of a relation is computed by getting the selectivities of individual select conditions. Selectivity of a select condition is computed by using information stored in the histogram of the attribute involved in the select condition. We are giving details about the information required to calculate selectivity, and we are not giving more details about selectivity computation process. Oracle 9i RDBMS uses width based histograms and stores them in a view DBA_ALL_TAB_HISTOGRAMS [6]. The view DBA_ALL_TAB_HISTOGRAMS stores frequency and normalized values of boundaries of the buckets. On the other hand DB2 7.0 RDBMS uses end-biased histograms and stores them in a table SYSIBM.COLDIST [12]. To keep track of selectivity of the range predicates it stores information about data distribution also in form of quantiles in the same table SYSIBM.COLDIST. Instead of storing normalized value of the attribute in histograms it stores actual values of the attribute. Different RDBMS support different data types, and selectivity estimation depends on data type of the attribute involved in the select condition. Because of these dependencies on RDBMS the function *setETS()* is implemented in the optimizer specific class.

The member function *setTS()* computes *TS* feature of the per table feature vector. The *TS* is the size of underlying the relation. It is computed as product of total size

[3]Refers to feature vector relevant to whole query.

[4]Refers to ratio of total size of a tuple of the relation to size of portion of the tuple required for the query execution.

of tuple and cardinality of the relation. Oracle 9i stores this information in the views SYS.DBA_TAB_COL_STATISTICS and SYS.DBA_TABLES [6]. DB2 7.0 stores this information in the tables SYSIBM.SYSTABLES and SYSIBM.SYSCOLUMNS [12]. To make this function independent of the optimizer we define some views on previously mentioned tables, so as to create same interface. Because of same interface the function *setTS()* is defined in the generic class *statCollect*.

Clustering Mechanism: The *leader* algorithm [4] is used to determine cluster representatives. To reduce search time we store clusters in tree structured schema.

Visual Interfaces: Using Java Swing, we have provided graphical interfaces to submit queries and observe the execution plans, to auto generate, re-organise and visualize clusters.

A sample interface is shown in Figure1 in appendix, which shows the plan generated by PLASTIC for a query7 in TPC-H benchmark.

Another sample interface is shown in Figure2 in appendix, which shows various clusters for a query5 of TPC-H benchmark, generated using *Auto Cluster Generator*.

Another sample interface is shown in Figure3 in appendix, which shows plan diagram for a query2 of TPC-H benchmark for oracle database, generated using the *Plan and Cost Diagram Generator*.

Another sample interface is shown in Figure4 in appendix, which shows cost diagram for a query2 of TPC-H benchmark for oracle database, generated using the *Plan and Cost Diagram Generator*.

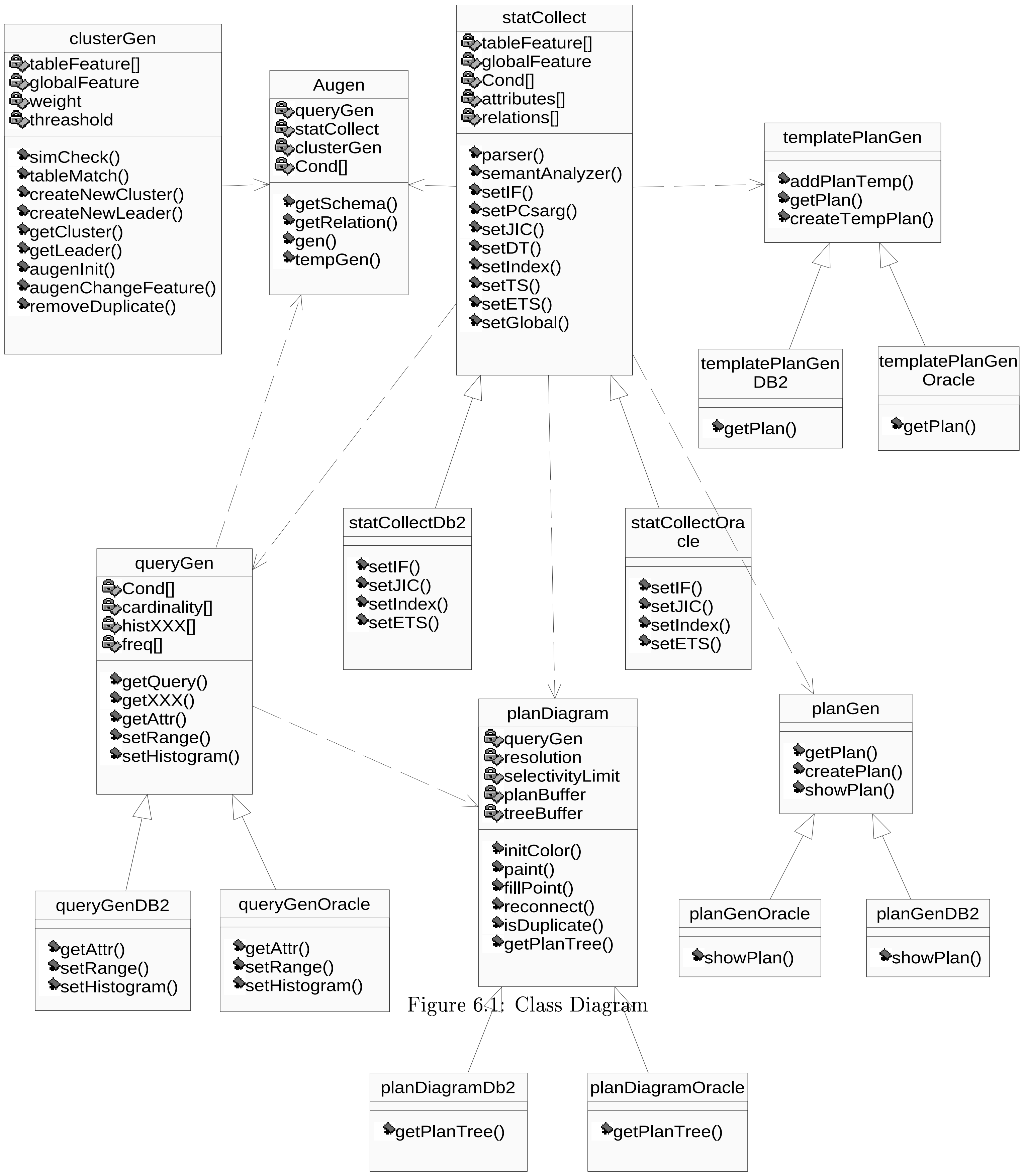


Figure 6.1: Class Diagram

File: H:\er.mdl 12:07:50 PM Tuesday, December 31, 2002 Class Diagram: Logical View / Main Page 1

Chapter 7

Improvements

In this chapter we are describing some improvements to feature vector and distance function of PLASTIC. And based on these improvements modified SIMCHECK algorithm.

7.1 Improvements to feature Vector

In this section we are describing some improvements to feature vector in PLASTIC to make it more versatile. Initially PLASTIC feature vector was restrictive and it was not taking into account things like system state, indexes and table organization mechanism. To make it less restrictive and exhaustive we have have removed, modified and added new features to feature vector.

Relation Access Path: One modification that we have done to the feature vector is to take different types of relation access paths [1] into account. There are different ways to access a relation for query execution. If there is a highly selective condition on a relation and there exists an index on a attribute involved in the condition, then in most of the cases the index scan is used to access the relation. Similarly if there is no highly selective condition on a relation then there are more chances of choosing the table scan. If we do mistake of choosing the index scan instead of the table scan or vice - versa then there may be big difference in the execution time between

[1] Refers to different ways of accessing underlying base table.

two plans. To distinguish between cases where the index scan is used and where the table scan can be used we have added one more feature *relSel* to the per table feature vector. It is selectivity of the most highly selective condition on a relation which can be executed using an index. If there is no condition on a relation, or condition can not be executed using indexes available on the relation, then it has value of *null* indicating index not available.

Commercial RDBMS support different types of indexes. For example DB2 has unique, cluster and reverse type of indexes [8]. To distinguish between different types of indexes we add one more feature *indexType* to the per table feature vector indicating index type.

Organization Mechanism: Relations can be stored using different types of organization mechanism. For example oracle9i has nested tables, partitioned table, B+Tree organized table etc. The organization mechanism also affects the query plan. To distinguish between different type of organization mechanism we have added one feature *tType* to the per table feature vector.

Feedback: Initially the feature vector used to be in the query space. It results in a strategy that if there is a resource then it will be used, but this may not be true. In some cases even if there is an index on a attribute involved in a condition on a relation optimized plan does not use the index because a large portion of the relation is being accessed. A cluster representative query is having an index on a relation and the index is not being used in the optimized plan, but still a in-coming query that is not having an index on the relation will not share plan with the cluster representative. To increase plan sharing, based on feedback from query plan for cluster representative, feature vector is made less restrictive. For example if the index on a relation is not being used in the optimized plan then *relSel* and *index flag* feature of the relation in the leader query are modified after getting the plan. This feedback process is done only for the cluster representative so it will not result in inefficient processing. For example consider a query on TPC-H benchmark where index is present for attribute p_partkey

```
select *
from
  part, partsupp
where
  p_partkey = ps_partkey
  and p_partkey > 3200
  and ps_supplycost > 400
```

But the plan for this query does not use the index so based on feedback value of*relSel* is changed to value of *null*. Now if a new query having relation part_m instead of part (only difference between part and part_m is that it is not having index on p_partkey) will reuse the plan of previous query.

System Parameter: System parameters like parallel processing factor and memory buffer size have a great impact on the query execution. Level of parallel processing that can be done in the query execution also affects the query plan. We have added one feature *pFactor* to distinguish between different level of parallel processing.

Based on above mentioned improvements new feature vector is given in table 7.1

All above mentioned improvement to feature vector make feature vector more exhaustive and increase sharing. Depending upon RDBMS query processing techniques some more feature can be added to feature vector.

Example Feature Vector: Consider simplified version of the query12 given in TPC-H benchmark where indexes are present for the attributes l_partkey and p_partkey of the tables LINEITEM and PART, respectively.

```
select
  p_type, l_extendedprice, l_discount
from
  lineitem, part
where
```

Feature	Description
	Global Features
NT	Number of tables participating in the query
DS	Degree sequence of the query
NJP	Total number of join predicates the query
$pFactor$	Amount of parallelism in the query execution
	Table Features
DT_i	Degree of table T_i
IF_i	Boolean indicating index-only access to T_i
TS_i	Size of T_i
ETS_i	(estimated) Effective size of T_i
$relSel_i$	Selectivity of most highly selective condition on T_i
$tType_i$	Type of table T_i

Table 7.1: Contents of a Query Feature Vector

l_partkey = p_partkey

and l_shipdate >= date ('1995-07-01')

The feature vector for this query is shown in Table 7.2. Note that the Index Flag is not set for both the tables since all the selection predicates and projections on the tables can not be evaluated through single common index. If we change p_type attribute in select predicate to p_partkey then *Index Flag* for table PART will be set. *pFactor* has a value of 10 indicating that 10 percent parallelism is possible in the query execution. *relSel* has null value for both the tables indicating that an index is not available.

Global Feature	Value
NT	2
DS	(1,1)
NJP	1
$pFactor$	10

Table Feature	Table PART	Table LINEITEM
DT_i	1	1
IF_i	0	0
TS_i	1014000000	34200000
ETS_i	113000000	800000
$relSel_i$	null	null
$tType_i$	1	1

Table 7.2: Example of a Query Feature Vector

7.2 Improvements to distance function

Original SIMCHECK algorithm[2] uses distance function $dist_{ij}(T_1^i,T_2^j)$, where T_1^i and T_2^j are the tables whose distance is computed, with T_1^i denoting i^{th} table of first query which is to be mapped with T_2^j, the j^{th} table of second query. The larger the distance lesser the similarity. In terms of statistical features given in Table 7.1, The distance function is given as:

$$dist_{ij}(T_1^i,T_2^j) = \frac{w_1 * |TS_1^i - TS_2^j| + w_2 * |ETS_1^i - ETS_2^j|}{max(TS_1^i,TS_2^j)}$$

Here, w1 and w2 are weighting factors (with w1 + w2 = 1). If w1 is given more value than w2 then it results in more emphasis on the base table size vice- versa. To increase sharing and accuracy we are suggesting three improvements to above mentioned distance function.

First Improvement: Originally, w1 is set higher than w2, which results in more emphasis on base table size. Queries across multiple schema will have different base table sizes. So setting w1 higher than w2 results in less sharing in queries across multiple schema. Based on experiments on queries across multiple schema and analysis of

[2]This algorithm is used to check similarity between two query feature vectors

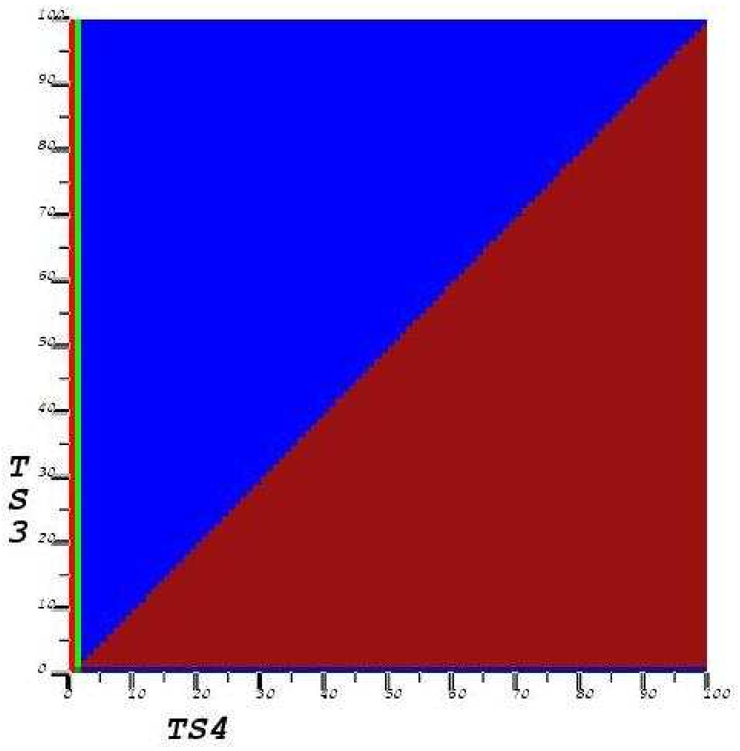


Figure 7.1: Plan Diagram No1

cost functions[3] we found that base table size effects the relation access path but it does not effect the join orders [4]. Vice - versa effective table size effects the join orders but it does not effect relation access path. And if base table size of a relation is increasing then plan does not change relation access path of the relation from the index scan to the table scan and vice - versa if base table size of a relation is decreasing then plan does not change relation access path of the relation from the index scan to the table scan.

For example Figure 7.1 and Figure 7.2 show plan diagram for two queries having same structure but different base tables. Base tables in the second query are ten times larger than base tables in the first query. It can be easily seen from the figures

[3]Refers to functions used to evaluate cost associated with different types of possible operations used to execute query.

[4]Refers to the order of joins between relations and join type.

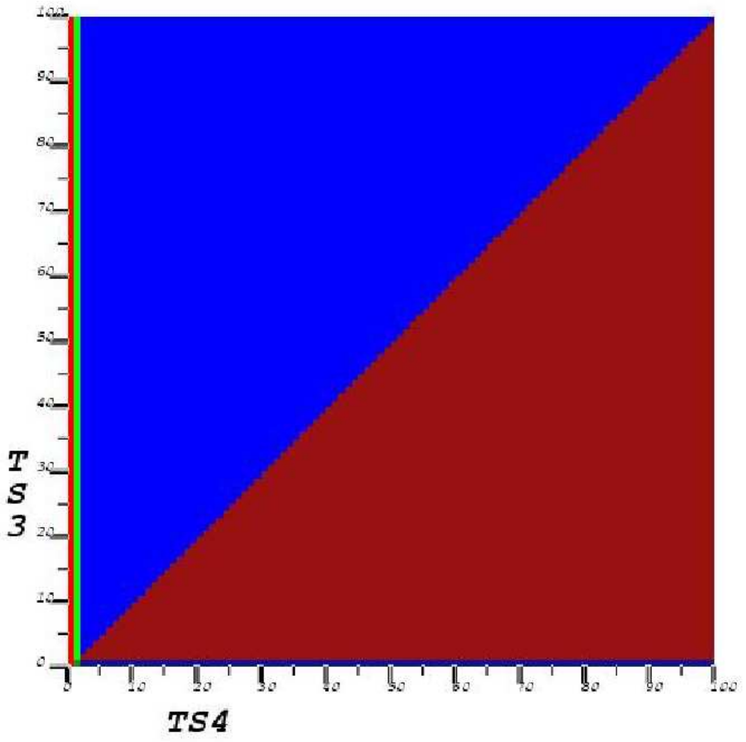


Figure 7.2: Plan Diagram No2

that the base table size has very less impact on query plans.

Based on previous the discussion we can say that base table size TS has very little impact on query plan. So we have remove TS component from the numerator of distance function and a separate check has been added for TS. And also in any of above mentioned conditions TS check is not done, otherwise it is done. Based on above mentioned change new distance function is:

$dist_{ij}(T_1^i,T_2^j) = \frac{|ETS_1^i - ETS_2^j|}{max(TS_1^i,TS_2^j)}$

Second Improvement: Originally, difference between table sizes is normalized by maximum of base table sizes. But as we mentioned previously base table size does not have effect on join orders, so in the new distance function function the difference between table sizes is normalized by maximum of effective table sizes. Based on above mentioned change new distance function is:

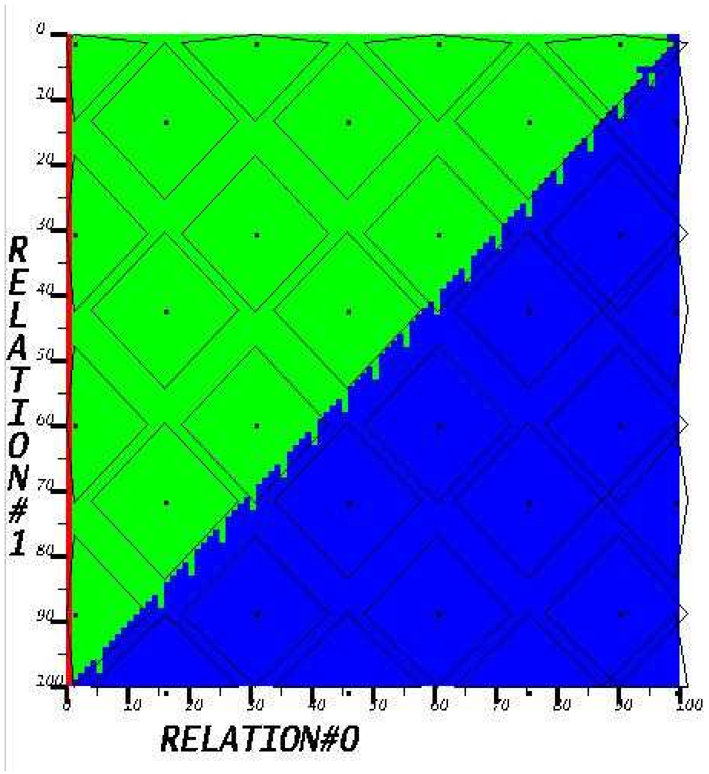


Figure 7.3: Plan Diagram For Fixed Sized Cluster

$$dist_{ij}(T_1^i, T_2^j) = \frac{|TS_1^i - TS_2^j|}{max(ETS_1^i, ETS_2^j)}$$

Third Improvement: In *SimCheck* function distance got from the distance function is compared against a threshold. So the size of a cluster dependents on the threshold. The larger the threshold bigger the size of the cluster. Originally, the distance is compared against a fixed threshold, which results in fixed sized clusters. The fixed sized clusters result in redundant clusters in the regions where plans do not change, and less clusters in the regions where plans change. Consider the plan diagram shown in Figure 7.3 for a query template

select *

from scott.ts1, scott.ts2

where ts1.c1 = ts2.c1

and ts1.c3 > :v1

and ts2.c3 > :v2

It can be easily seen that there are many clusters in the regions having less plans, and there are less clusters in the region having more plans. Solution is to have big clusters in the regions where plans do not change and small clusters in the regions where plans change. Based on experiments and analysis of the cost functions we found that there are more plans in the regions where effective table sizes of the relations involved in a query are close to each other vice - versa. So there should be small clusters in the regions where effective table sizes are close to each other. To exploit this property of plan diagrams, threshold is computed as variance in effective table sizes of tables involved in a query.

Now we are describing the process of threshold computation.

First, tables are grouped based on standard deviation of effective table sizes in query. If standard deviation in a group is less than some fixed value[5] then threshold is calculated as $\frac{deviation}{(numberoftablesingroup)^2}$, otherwise the group is broken into parts process continues recursively. The pseudo code of algorithm *THCALC* is given in figure, takes as input effective table size of tables involved in query.

So now clusters having effective table sizes close to each other will have less threshold and vice versa. This results in small sized clusters in the regions having more number of plans, and big sized clusters in the regions having less number of plans. Variable

[5]Value of 0.7 results in good groups

```
SIMCHECK (Effective table sizes)
1.   Calculate standard deviation $ETSSD$ of effective table size of tables
2.   IF $ETSSD > FValue$
3.   Break group in two parts g1,g2 and call algorithm for each group
4.   ELSE IF Number of tables$_{group}$ = 1
       $thresholdETS_{group} = FXTHOLD$
     ELSE
       $thresholdETS_{group} = \frac{Variance_{group}}{(Numberoftables_{group})^2}$
```

Figure 7.4: **The THCALC Algorithm**

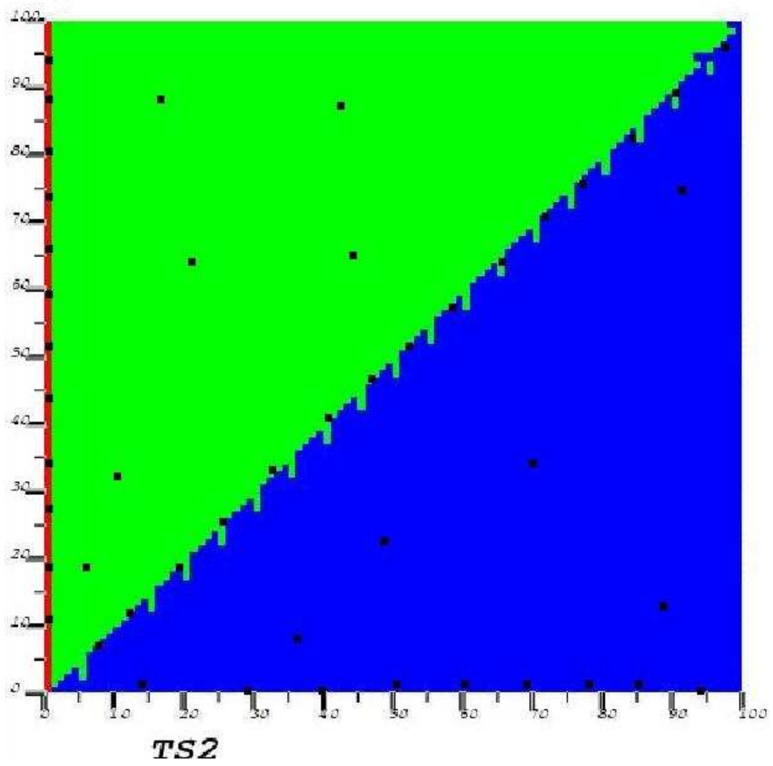


Figure 7.5: Plan Diagram For Variable Sized Cluster

sized clustering results in following benefits.

First, it results in less number of clusters which increases efficiency. Second, it results in more clusters in the regions having more plans which reduces chance of errors [6] and risk [7].

For example in Figure 7.5 shows plan diagram for a query template

select *

from scott.ts1, scott.ts2

where ts1.c1 = ts2.c1

and ts1.c3 > :v1

and ts2.c3 > :v2

[6]It is ability of the cluster boundaries to discriminate the plan boundaries sketched by an optimizer

[7]This is computed as the worst case extra cost incurred when PLASTIC does not choose the optimizer-generated plan for a particular query

It can be seen that there are many clusters in the regions having more plans, and less clusters in the regions having less plans. Above mentioned improvements result in variable sized oval shaped clusters which are difficult to show, so we have shown only the cluster representatives.

All three mentioned improvements result in increased sharing across multiple schema, increased efficiency and more accurate plan assignments. Note that this threshold computation process is done only for cluster representatives so it will not result in more classification time.

7.3 New Distance Function

Based on improvements done in previous chapter pseudo code of modified SIMCHECK algorithm is given in figure. This algorithm takes two query features and outputs a boolean value indicating whether or not they are similar. The algorithm operates in two phases, structural map which mainly consist of boolean checks and statistical map which consist of mapping of the tables using distance function. In the first phase, the feature vector are compared for equality on the number of tables, the sum of table degrees and parallel processing factor. Only if feature vector match in first phase, the second phase is invoked, otherwise the queries are deemed to be dis-similar. The structural map is done first in order to identify dissimilar queries as early and as simply as possible. For example, it is obvious that if the number of tables in the queries do not match, then their plan will also necessary have to be different. Such structural map are done at early phase as an effective mechanism for stopping unproductive match at early stage.

```
SIMCHECK ($Q1, Q2$)
  // Check that Queries have same number of Tables
1. If $Q1.NT$ != $Q2.NT$ RETURN (Not Similar);
  // Match Query Level Semantics
2. If $DS(Q1) = DS(Q2)$ AND
     $NJP(Q1) = NJP(Q2)$ AND
     $pFactor(Q1) = pFactor(Q2)$ AND
          GO TO Step 4;
3. RETURN (Not Similar);
  //—— Find the Best Mapping between Tables ——
4. For every group g of tables
// all the tables inside the group are in sorted order of effective tables sizes
       $R_1 = T_1, T_2,.., T_k \quad R_1 \subseteq Q_1$
       $R_2 = T'_1, T'_2,...,T'_k \quad R_2 \subseteq Q_2$
      find the mappings of compatible tables between $R_1$ and $R_2$
      that has the minimum distance, with
      respect to pairwise distance function
      If IF($T_1^i$)=IF($T_2^j$) AND
      tType($T_1^i$)=tType($T_2^j$) AND
      DT($T_1^i$)=DT($T_2^j$) AND
      $\frac{|TS_1^i - TS_2^j|}{max(TS_1^i, TS_2^j)} < ThresholdTS$[a]
      then
       $dist_{ij}(T_1^i, T_2^j) = \frac{|ETS_1^i - ETS_2^j|}{max(ETS_1^i, ETS_2^j)}$
      else
       $dist_{ij}(T_1^i, T_2^j) = 1.0$
          Compute the difference, seldist, of relSel
      between compatible tables $T_i, T_j$ using function
      $seldist_{i,j}(T_1^i, T_2^j) = |relSel_1^i - relSel_2^j|$
6.  // compare the distance against group threshold
      IF $dist_{ij}(T_1^i, T_2^j) > ThresholdETS_g$ OR
       $seldist_{i,j}(T_1^i, T_2^j) > ThresholdETS_g$
      RETURN (Not Similar);
7.RETURN (Similar);
```

[a]Threshold used for base table size comparision, value of 0.01 gives accurate results.

Figure 7.6: **The Modified SIMCHECK Algorithm**

Chapter 8

Future Work

The future avenues include extending the work to handle nested queries, groups and aggregates. Our preliminary experience with 'group by' suggests that it should be simple to apply SIMCHECK Algorithm for group-bys Figure7.6. This is because in most of the cases group-by works on the output given by the basic query.

Bibliography

[1] V. Sengar and J. Haritsa "Plan Recycling through Incremental Clustering", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2003.

[2] 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 (VLDB)*, August 2002.

[3] http://www.cs.toronto.edu/ jglu/home/tools.htm

[4] J. Hartigan, *Clustering Algorithms*, John Wiley & Sons, Inc., 1975.

[5] http://www.tpc.org

[6] *http://download-east.oracle.com/otndoc/oracle9i/toc.htm*

[7] *http://www.va.pubnix.com/man/xdb/sqlref/ SystemTableDescriptions_434.html*

[8] *http://www.craigsmullins.com/db2_type.htm*

[9] K. Shim, T. Sellis and D. Nau, "Improvements on a heuristic algorithm for multiple-query optimization", *Data and Knowledge Engineering*, 12, 1994.

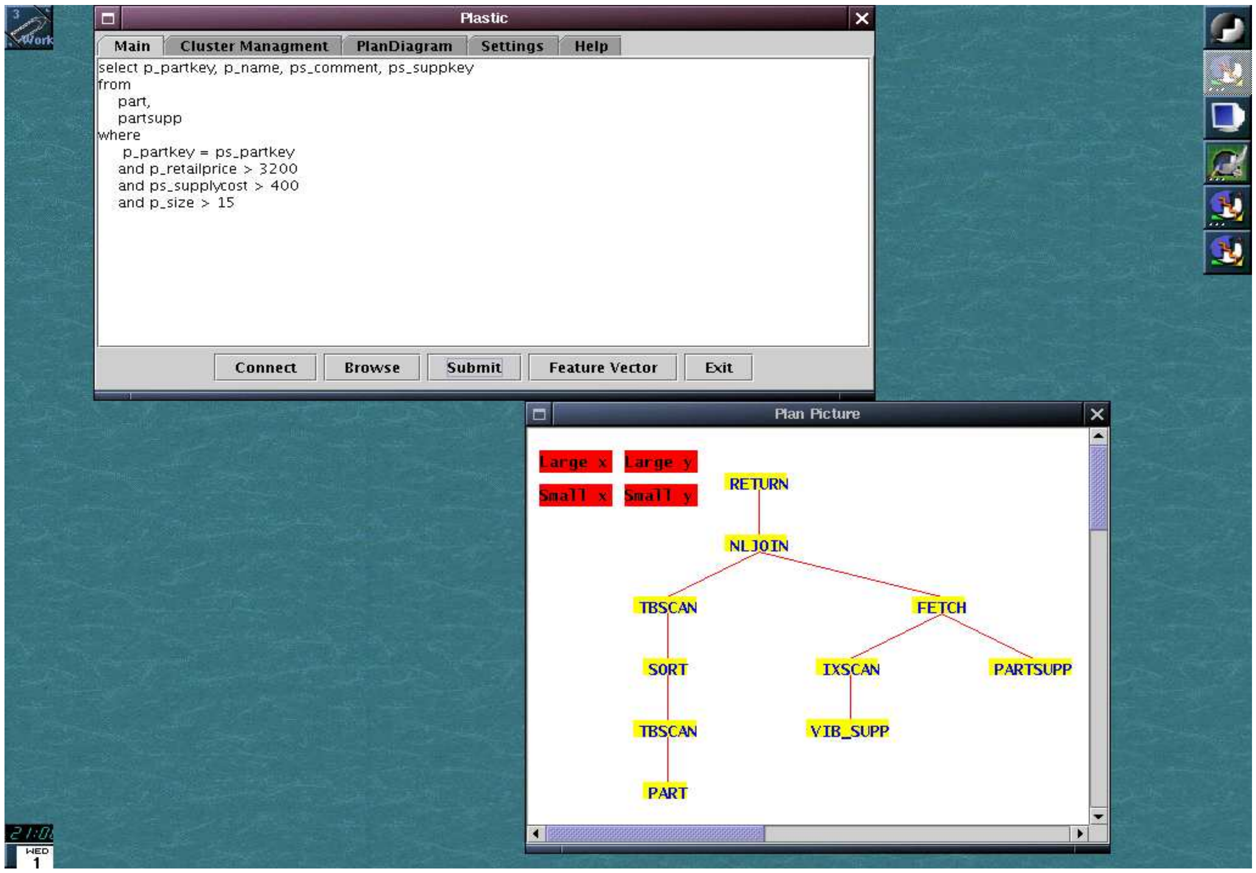
[10] *http://www.sql.org.*

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

[12] P. Gassner, G. Lohman, K. Schiefer and Y. Wang, “Query Optimization in the IBM DB2 Family”, *Data Engineering Bulletin*, 16 (4), (1993).

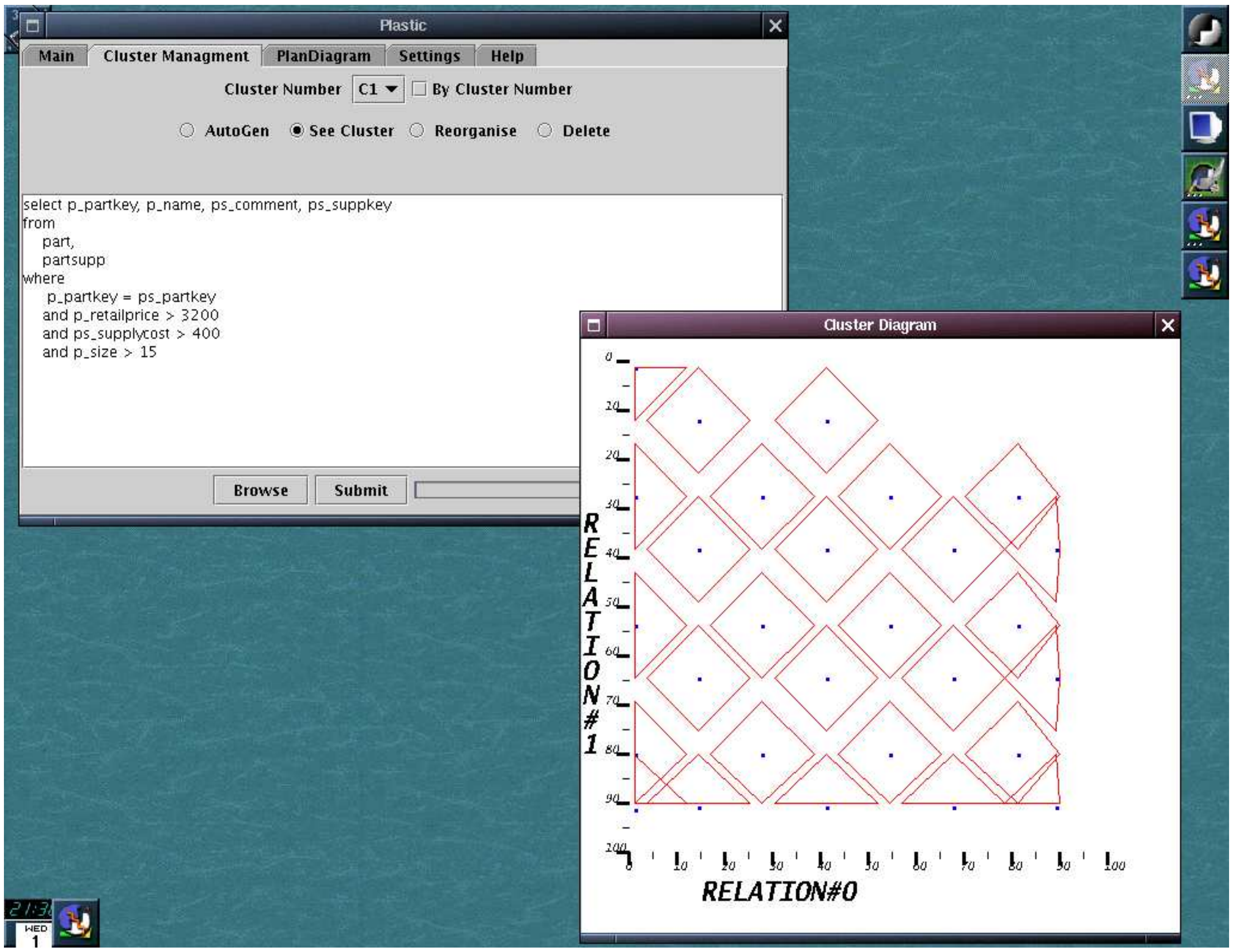
Appendix A

Screenshots of PLASTIC



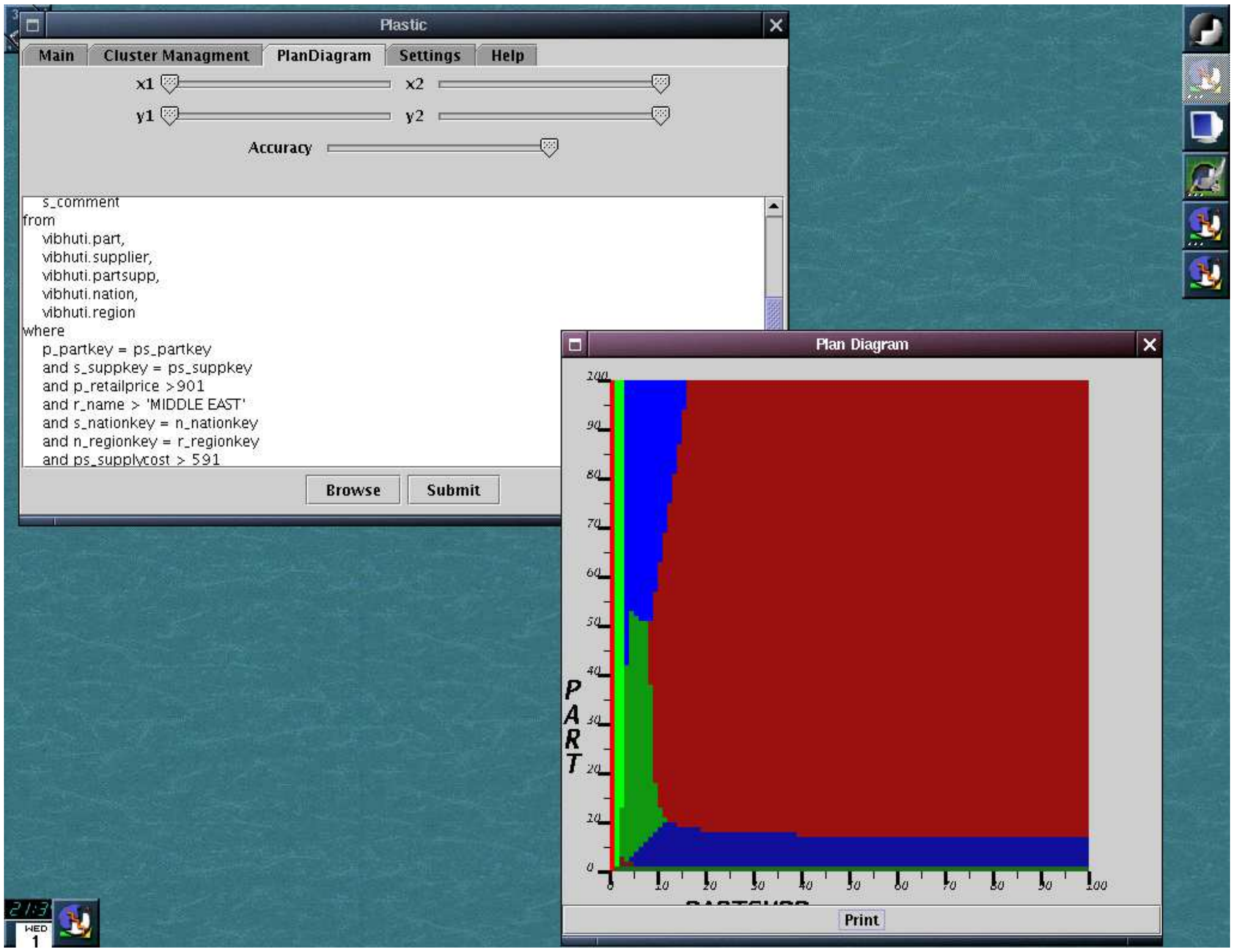
```
select p_partkey, p_name, ps_comment, ps_suppkey
from
   part,
   partsupp
where
    p_partkey = ps_partkey
   and p_retailprice > 3200
   and ps_supplycost > 400
   and p_size > 15
```

Figure A.1: Plan Interface



```
select p_partkey, p_name, ps_comment, ps_suppkey
from
    part,
    partsupp
where
     p_partkey = ps_partkey
    and p_retailprice > 3200
    and ps_supplycost > 400
    and p_size > 15
```

Figure A.2: Clusters Interface



```
   s_comment
from
   vibhuti.part,
   vibhuti.supplier,
   vibhuti.partsupp,
   vibhuti.nation,
   vibhuti.region
where
   p_partkey = ps_partkey
   and s_suppkey = ps_suppkey
   and p_retailprice >901
   and r_name > 'MIDDLE EAST'
   and s_nationkey = n_nationkey
   and n_regionkey = r_regionkey
   and ps_supplycost > 591
```

Figure A.3: Plan Diagram Interface