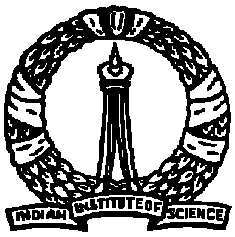
Green Query Optimization: Taming Query Optimization Overheads through Plan Recycling

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

by

Sarda Parag Kacharulal



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

JULY 2004

to Pradip Bhaiyya

without whose constant support and urging
I would never have made it this far.

Acknowledgments

I would like to begin by thanking my guide Prof. Jayant Haritsa for his enduring support and encouragement as well as the freedom to pursue my area of interest. I would also like to thank my fellow ME-DSL'ites (in alphabetical order) Abhijit, Amol, Anoop and Shipra for their wonderful company and help. Last but not least, I thank all my friends who directly or indirectly helped me to complete this report.

Abstract

PLASTIC [5] is a recently-proposed tool to help query optimizers significantly amortize optimization overheads through a technique of plan recycling. 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. We have now significantly extended the scope, useability, and efficiency of PLASTIC, by incorporating a variety of new features, including an enhanced query feature vector, variable-sized clustering and a decision-tree-based query classifier. This tool is working on the commercial database platforms (Oracle 9i and IBM DB2).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Query optimization is well-known to be a computationally intensive process since a combinatorially large set of alternatives have to be considered and evaluated in order to find an efficient access plan [17]. 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 [23]. These well-known inherent costs of query optimization are compounded by the fact that a query submitted to the database system is typically optimized afresh, providing no opportunity to amortize these overheads over prior optimizations. While current commercial query optimizers do provide facilities for reusing execution plans (e.g. "stored outlines" in Oracle 9i [24]), the query matching is extremely restrictive – only if the incoming query has a close *textual resemblance* with one of the stored queries then the associated plan is re-used to execute the incoming query.

The recently-proposed PLASTIC tool [5] attempts to significantly increase the scope of plan reuse. It is based on the observation that even queries which differ in projection, selection and join predicates, as also in the base tables themselves, may still have identical *plan templates* – that is, they share a common database operator tree, although the specific inputs to these operators could be different. By identifying such similarities in the plan space, the utility of plan cacheing can be materially improved.

PLASTIC captures these similarities by characterizing queries in terms of a feature vector that includes structural attributes such as the number of tables and joins in the query, as well as statistical quantities such as the sizes of the tables participating in the query. Using a distance function defined on these feature vectors, queries are grouped into clusters. Clusters are build incrementally using *leader* algorithm proposed by Hartinger [8]. Each cluster has a representative for whom the template of the

optimizer-generated execution plan is persistently stored, and this plan template is used to execute all future queries assigned to the cluster. In short, PLASTIC recycles plan templates based on the expectation that its clustering mechanism is likely to assign an execution plan identical to what the optimizer would have produced on the same query. Experiments with a variety of TPC-H-based queries on a commercial optimizer showed 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 because optimizers can now always run at their highest optimization level – the cost of such optimization is amortized over all future queries that reuse these plans. Further, 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 assignment of queries to clusters is based on database statistics, the plan choice for a given query is *adaptive* to the current state of the database.

A basic prototype implementation of PLASTIC on commercial database systems was developed [16] and it will be refer as Version-1 system hereafter. We have significantly extended the scope, useability, and efficiency of PLASTIC and added a host of new features that support these extensions. These new features which are now part of Version-2 system include:

- augmented feature vector to account for table access paths and organizations, index types;
- variable-sized clustering to match volatility in plan space;
- integration of a classical C5.0 [25] decision tree classifier for fast cluster identification;
- a plan analyzer module for comparing plan choices within and across database platforms;
- computation and incorporation of cost estimations in plan generation; and
- automated mechanisms for creating and visualizing integrated "plan-cost diagrams", which enumerate the plans chosen by the optimizer over the query space, and show the associated costs.

The remainder of this report is organized as follows: PLASTIC Version-1 system is discussed in Chapter 2 and the new features of PLASTIC Version-2 system are discussed in Chapter 3. In Chapter 4, we discuss implementation specific issues. The experimental setup and results obtained are presented in Chapter 5. We conclude with a summary in Chapter 6.

Chapter 2

PLASTIC Version-1 System

In this chapter, we describe PLASTIC Version-1 system: *already developed*, a basic prototype implementation of PLASTIC on commercial database systems [16]. This chapter is organized as follows: In Section 2.1, we present an overview on block level architecture of the PLASTIC. The feature vector used to represent the query and the algorithm for determining query similarity are discussed in Section 2.2 and Section 2.3 respectively[1].

2.1 Architecture of PLASTIC

A block-level diagram of the PLASTIC architecture is shown in Figure 2.1. The user query is first processed by the *Feature Vector Extractor* which also accesses the system catalogs and obtains the information required to produce the feature vector. The *SimilarityCheck* module establishes whether this feature vector has a sufficiently close match with any of the cluster representatives in the *Query Cluster Database*. If a match is found (solid lines in Figure 2.1), the plan template for the matching cluster representative is accessed from the *Plan Template Database*. The plan template is converted into a complete plan by the *Plan Generator* module, which fills in the operator inputs based on the specifics of the current query.

On the other hand, if no matching cluster is found (dashed lines in Figure 2.1), the Query Optimizer is invoked in the traditional manner and its output plan is used to execute the query. This plan is also passed to the *Plan Template Generator* which converts the plan into its template representation for storage in the *Plan Template Database*. Concurrently, the query feature vector is stored in the *Query Cluster Database*,

[1]The description of PLASTIC Version-1 system appeared in this chapter is taken from [5]

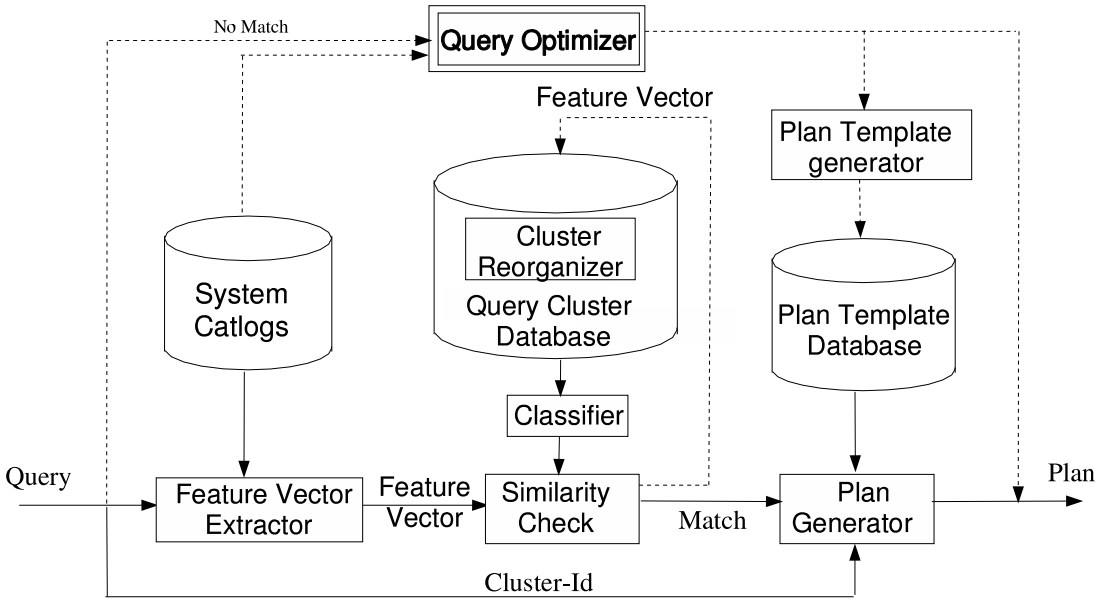


Figure 2.1: The PLASTIC Architecture

as a new cluster representative. Periodically, the cluster database may be reorganized to suit constraints such as a memory budget or a ceiling on the the number of clusters.

2.2 Query Representation

We will discuss, with reference to PLASTIC Version-1 system, the issue of query representation by its feature vector since the appropriate choice of features forms the core of any clustering approach.

We will use Figure 2.2 to motivate and explain some of the features. This figure shows two graphs which represent two different queries on six tables each. In these graphs, the nodes A, B, ..., F and P, Q, ..., U represent the tables and the lines between them represent the *join* predicates relating them. The features used by Version-1 system are the following:

Number of Table (NT): This is the number of tables participating in query.

Number of Table (NJP): This is the number of join predicates present in query.

Degree of a Table (DT): This is the number of *join* predicates in which a particular table is involved. For example, the degree of table E in Figure 2.2(a) is 3. This feature plays a role in positioning the table within the join tree in the access plan of the corresponding query.

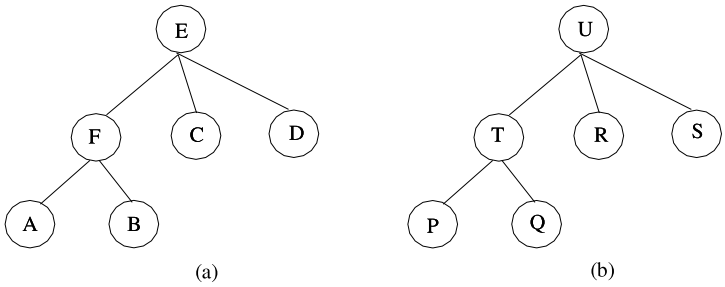


Figure 2.2: Degree Graphs

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 2.2 is (3, 3, 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.

Predicate Counts of a Table (PC): A predicate can be, as per the definition in the System R optimizer [13], 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, *name like 'abc%'* is an SARGable predicate while *name like '%abc'* is not. For each table involved in the query and for each predicate type, the count of the number of such predicates operating on the table is maintained.

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.

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 the effective size of each table participating in a join. It is calculated by estimating, through the statistics present in the system catalogs, the impact of pushing down all the projections and selections on this table that appear in the query.

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
	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 2.1: Query Feature Vector (PLASTIC Version-1 System)

Putting all of the above together, the complete query feature vector definition is as shown in Table 2.1. For ease of understanding, features has been separated into *Global Features*, which are query-wide values, and *Table Features*, which are relevant to each individual tables. This feature vector is illustrated by an example below.

Example 1 Consider the query

```
select A.a1, B.b2
from A, B
where A.a1 = B.b1
```

where indexes are present for the attributes $a1$ and $b1$ of tables A and B, respectively. The feature vector for this query is shown in Table 2.2. Note here that the index flag is set for table A since all attributes related to A – in this case, $a1$ – are accessible through the index. However, this flag is not set for table B since the $b2$ projection attribute is not accessible through the $b1$ index. But, if we had happened to have a multi-attribute index $(b1, b2)$ on table B, then the index flag would also have been set true for B. PLASTIC do not estimate selectivities for join predicates because for that a join order should be apriori known and this information is only available from the optimization process. Since, only projections of the tables A and B are taking part in join, ETS values for these tables are not the same as their TS values. □

Global Feature	Value
NT	2
DS	(1,1)
NJP	1
JIC	{0,0,1}

Table Feature	Table A	Table B
DT_i	1	1
IF_i	1	0
$PCsarg_i$	1	1
$PCnsarg_i$	0	0
JIC_i	{0,0,1}	{0,0,1}
TS_i	200000	100000
ETS_i	20000	50000

Table 2.2: Example Query Feature Vector

2.3 Establishing Similarity

We now move on to the next issue of determining when two queries are said to be similar. Given the 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.

To avoid the problem, PLASTIC takes a different approach. That is, it tries to establish a notion of similarity that facilitates both (a) efficient classification of new queries based on features only, and (b) that the plan chosen by the optimizer for the queries within a cluster is the same in the majority of the cases. This approach is referred to as the SIMCHECK algorithm and is described below.

2.3.1 The SIMCHECK Algorithm

The SIMCHECK algorithm, whose pseudo-code is shown in Figure 2.3, takes as input two query feature vectors and outputs a boolean value indicating whether they are similar or not. The algorithm operates in two phases, "Feature Vector Comparisons" and "Mapping Tables". In the first phase, the feature vectors are compare for equality on the number of tables, number of join predicates and table degrees. 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 implementation,

INPUT : Query feature vectors Q_1, Q_2
OUTPUT: boolean value indicating whether Q_1, Q_2 are similar or not

SIMCHECK (Q_1, Q_2)
{
// Check that Queries have same number of Tables
1. IF NT(Q_1) != NT(Q_2) RETURN (Not Similar);

// Check that Queries have Level Semantics
2. IF DS(Q_1) != DS(Q_2) OR NJP(Q_1) != NJP(Q_2) RETURN (Not Similar);

// Find the Best Mapping between Tables
4. Find the best one to one matching between compatible tables of Q_1, Q_2 and let
$R_1 = T_1^1, T_1^2 \ldots T_1^K$ are tables in Q_1
$R_2 = T_2^1, T_2^2 \ldots T_2^K$ are tables in Q_2
such that T_1^i is mapped with T_2^i $(1 \leq i \leq k)$

5. IF $\sum_{i=1}^{K} \frac{|TS_1^i - TS_2^i|}{max(TS_1^i, TS_2^i)} \geq$ TS-Threshold RETURN (Not Similar);

6. IF $\sum_{i=1}^{K} \frac{|ETS_1^i - ETS_2^i|}{max(TS_1^i, TS_2^i)} \geq$ ETS-Threshold RETURN (Not Similar);

7. RETURN (Similar);
}

Figure 2.3: The SIMCHECK Algorithm

cluster representatives are grouped based on these zero-in equality conditions and these conditions are evaluated against groups rather than each cluster representative.

In the Mapping Tables phase, SIMCHECK 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 plan for accessing the mapped tables. The two tables are said to be compatible if they have same degrees, join index counts and index flag. In the first step, the sets of *compatible* tables are determined and in second step, for every possible pair of compatible tables, tables are mapped based of original and (estimated) effective sizes so that difference between mapped tables would be minimal. Finally, the distance function along with the threshold value, which is algorithmic parameter, are used to test queries for similarity. The distance function captures the effect of effective table sizes of participating tables and is discussed next.

2.3.2 The Query Distance Function

After compatible tables are identified, SIMCHECK tries to establish valid one-to-one mappings. These mappings are then compared using their estimated effective sizes, through a distance function. The larger the distance, the lesser the similarity. The distance function used by PLASTIC Version-1 system is

$$Distance(Q_1, Q_2) = \sum_{i=1}^{K} \frac{\mid ETS_1^i - ETS_2^i \mid}{max(TS_1^i, TS_2^i)} \tag{2.1}$$

where Q_1 and Q_2 are queries involving K tables and TS_1^i, ETS_1^i is original and (estimated) effective size of i^{th} table in Q_1. Similar explanation holds for TS_2^i, ETS_2^i.

The distance function essentially computes the separation between two tables in terms of their ETS values with respect to the query. When there are multiple mappings possible between the tables of two queries, the mapping with the minimum distance must be chosen. Note that in distance function, numerator is normalized by the maximum of the sizes of the two relations. Thus the distances of all the table pairs for a particular mapping, is bounded above by the total number of tables in the query. Another feature of the distance function is that it is symmetrical, as can be readily seen from the formula.

Chapter 3

PLASTIC Version-2 System

In this chapter, we describe in detail the new features of PLASTIC Version-2 system. This chapter is organized as follows: In Section 3.1, we propose augmentation into feature vector, which is used to represent the query. The variable-sized clustering along with useful formulae related to it are discussed in Section 3.2. A decision tree based classifier is presented in Section 3.3 as a solution to the problem of efficiently determining matching cluster. A cost estimator module to estimate the cost of plan for matched query and plan-cost diagram generator are described in Section 3.4 and Section 3.5 respectively. In Section 3.6, we present an algorithm for comparing plan choices.

3.1 Augmenting Feature Vector

In this section, we propose some modifications into features to account for table access paths and organizations, index types and join operation. As mentioned in Chapter 1, PLASTIC is based on the expectation that its clustering mechanism is likely to group those queries together for whom the optimizer will generate same plan template. The plan chosen by optimizer is based on three factors – first is system settings like parallelism available, buffers allocated to sorting/hashing etc; the second is query structure which includes number of tables in query, join degree of each of tables, joining conditions, selectivities of tables etc. and the third is actual relations participating in query and indexes available on these relations.

System settings are set when database is created and since they are not susceptible to changes afterwords and even if they are changed, changes are not frequent one, PLASTIC tries to capture only last two factors in term of features. Note that, if settings are changed, then PLASTIC will need *only* to get new plan for all cluster representatives.

Since there are no confirmative rules determining plan choices and even all commercial optimizer systems selects the plan by enumerating the plan space, we are not giving theoretical proof for completeness of feature vector. We will rather describe how feature vector can be augmented so that PLASTIC captures almost all the factors affecting table access path/join method selection. For each of these methods, we will discuss the heuristics about when the specific method is used and then describe how PLASTIC can capture the factors included in these heuristics.

3.1.1 Table Access Path

- **Full Table Scans:** During a full table scan, all blocks in the table are scanned. Each row is examined to determine whether it satisfies the query's WHERE clause. The optimizer uses a full table scan in any of the following cases:

    1. Lack of Other Access Paths: If the query is unable to use any of existing indexes, then it uses a full table scan. To capture this factor, we propose to add SEL (selectivity) as feature for each of participating table. It will store minimum fraction of the table needs to be accessed by index available on it. If no index is available on particular table, SEL will be set to 1 for that table.
    2. Large Amount of Data: If the optimizer thinks that the query will access most of the blocks in the table, then it uses a full table scan, even though indexes might be available. Again, PLASTIC will be able capture this factor by SEL feature vector having the value near to 1 (although not 1).
    3. Small Table: If a table contains blocks less than some threshold[1] under the high water mark, which can be read in a single I/O call, then a full table scan might be cheaper than an index range scan, regardless of the fraction of tables being accessed or indexes present. The optimizer's decision to use full table scans is influenced by the percentage of blocks accessed, not rows. Since, for two tables having same logical size (cardinality and average length of tuple) optimizer may choose different access path, we propose to add BLOCKS as feature. It will store number of blocks in each of the participating table under high water mark.

- **Index Unique Scans**: Optimizer performs a unique scan if query contains a UNIQUE or a PRIMARY KEY constraint which guarantees only a single row is accessed i.e., all columns of a

[1] DB_FILE_MULTIBLOCK_READ_COUNT in Oracle

unique (B-tree) index are specified with equality conditions. We propose to add another feature, INXTYPE, determining type of index available on table. Index type could be unique index or clustered index or normal index or combination of these.

- **Index Range Scans**: The optimizer uses a range scan when it finds one or more leading columns of an index are specified in SARGable predicate and fraction of data to be accessed is low. PLASTIC determines the index available on relation having same leading columns as specified in SARGable predicate and fraction of data to be accessed is captured in SEL feature.

- **Fast Full Index Scans**: Fast full index scans are used when the index contains all the columns that are needed for the query, and at least one column in the index key has the NOT NULL constraint. A fast full scan accesses the data in the index itself, without accessing the base table. PLASTIC capture the factor by IF feature. IF is set to true if relation has a index that contains all columns mentioned in query.

- **Cluster Scans**: A cluster scan is used to retrieve, from a table stored in an indexed cluster, all rows that have the same cluster key value. In an indexed cluster, all rows with the same cluster key value are stored in the same data block. PLASTIC will be able to capture the factor by INXTYPE feature.

- **Hash Scans**: A hash scan is used to locate rows in a hash cluster, based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data block. We propose to set INXTYPE to HASH if table is stored in hash cluster.

3.1.2 Join Methods

- **Nested Loop Joins**: Nested loop joins are useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the inner table. In PLASTIC, the size of data being joined is captured in ETS feature and whether join is efficient way of accessing table or not is captured by JIC feature.

- **Hash Joins**: Hash joins are used for joining large data sets. The optimizer uses a hash join to join two tables if they are joined using an equijoin and if either of the following conditions are true: a large amount of data needs to be joined or a large fraction of the table needs to be joined. Size of dataset is captured in ETS feature. To consider whether conditions is equijoin or not, we propose

to add EJC (Equi-Join condition Count) feature. It will store, for each of participating relation, counts of equijoin-join conditions operating on that relation.

- **Sort Merge Joins**: Sort merge joins are useful when join condition between two tables is inequality condition like $<$, $\leq$, $>$, or $\geq$. Inequality conditions will be captured in DT, EJC features. Sort merge joins can perform better than hash joins if both of the row sources are sorted already or optimizer thinks that the cost of a hash join is higher, based on the buffers allocation parameters[2]. Size of data sets is captured by ETS feature and PLASTIC assumes that system settings are not changed.

- **Cartesian Joins**: Cartesian join is used when one or more of the tables does not have any join conditions to any other tables in the query. PLASTIC will be capture this factor by having degree of the specific table as 0.

Since all SARGable and non-SARGable predicates are applied in only one filter step as soon as table is scanned, the *exact count* predicates do not help in plan selection and therefore we propose to delete them from feature vector.

In summary, we propose to add four table features into PLASTIC namely, SEL (selectivity), BLOCKS (blocks under high water mark), INXTYPE (type of index) and EJC (Equi-Join conditions Count) and delete count of SARGable and non-SARGable predicate from features. The complete set of *new* feature vector for query in Example 1 is shown in Table 3.1.

Global Feature	Value
NT	2
NJP	1
DS	(1,1)
JIC	{0,0,1}

Table Feature	Table A	Table B
DT_i	1	1
EJC_i	1	1
JIC_i	{0,0,1}	{0,0,1}
IF_i	1	0
SEL_i	1	1
$INXTYPE_i$	NORMAL	NORMAL
$BLOCKS_i$	2000	1000
TS_i	200000	100000
ETS_i	20000	50000

Table 3.1: New Query Feature Vector

[2]HASH_AREA_SIZE and SORT_AREA_SIZE in case of Oracle and SORTHEAP, DBHEAP in case of DB2

3.2 Variable-Sized Clustering

A serious problem with PLASTIC Version-1 system's clustering mechanism is that, by virtue of distance function (Equation 2.1), clusters are *fixed-sized* – an example corresponding to an SPJ[3] version of Query 2 from the TPC-H benchmark [23] is shown in Figure 3.1 (the axes represents the selectivities of a pair of the participating relations, the dots represent the cluster representatives and background shows associated plan diagram [5].)

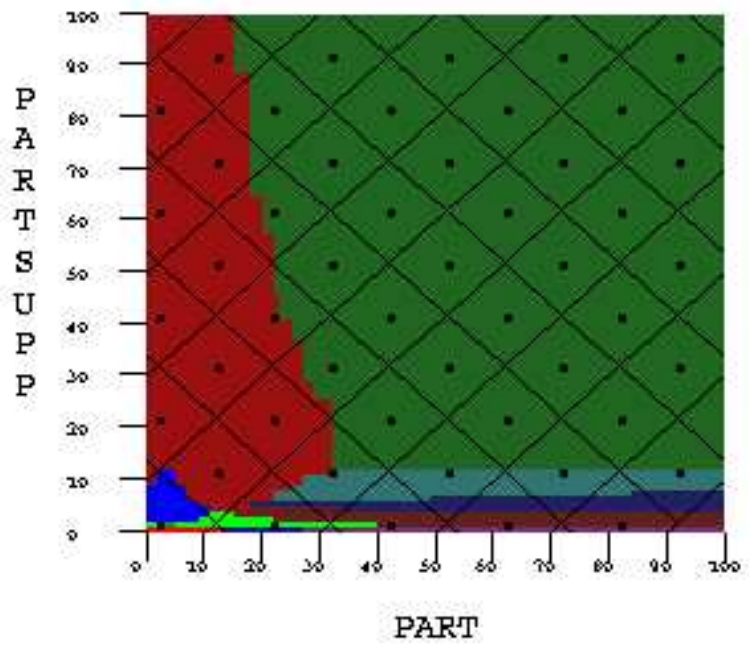


Figure 3.1: Fixed-Sized clustering

Specifically, the distance function in PLASTIC Version-1 system uses *manhattan* distance metric and produces clusters that are rhombus(diamond) in plan space with the stretch of any cluster, from it's representative, in all dimensions of this space is equal to threshold. Note that, even though denominator (TS value) for distance function changes along with dimension, the numerator (ETS value) also changes proportionally. So distance between two versions of same query is same as sum of absolute difference between selectivities of participating relations.

From the Figure 3.1, we can see that highly selective region of relations *PARTSUPP* is highly volatile (rapid changes in plan choices) and fixed-sized clusters are not able to cover it properly. Indeed, most

[3]Select-Project-Join

of clusters are wasted in covering region having same plan – one present at low selective (top right) region. In summary, the *fixed-size* clusters results in the twin problems of insufficient clusters in the *high-volatility* regions of the plan space and redundant clusters in the *low-volatility* regions. We have now incorporated *variable-sized* clustering in PLASTIC, providing several small-sized clusters in the high-volatility region and a few large-sized clusters in the low-volatility region.

3.2.1 Improvements

Our scheme is based on the observation that the high-volatility region is typically present in the highly-selective region of the plan space. Therefore, we have modified the distance function, used to compare similarities between queries, such that the cluster size thresholds are small in the highly-selective regions, and large in the less-selective regions. Highly selective condition on table results in lower ETS (Effective Table Size) value. We have used this value as denominator in the distance function (Equation 2.1). The modified distance function is

$$Distance(Q_1, Q_2) = \sum_{i=1}^{K} \frac{\mid ETS_1^i - ETS_2^i \mid}{max(ETS_1^i, ETS_2^i)} \tag{3.1}$$

We will now discuss some of the properties of new distance function. With the modified the distance function, queries with highly selective conditions will be deemed to be dis-similar even on small change in selectivity as opposed to queries with low selective conditions which will be dis-similar only on large change in selectivity. Note that in this distance function, numerator is normalized by the maximum of the effective sizes of the two relations. Thus, as in old distance function, the distances between two queries is bounded above by the total number of tables in the query. Similarly, new distance function is also symmetrical, as can be readily seen from the formula. The shape of clusters formed by this distance metric will not be *exact* rhombus but will be similar to it, as illustrated below.

Consider a cluster with representative at ETS value of X and having stretch from Y to Z of ETS value for same table as shown in Figure 3.2. Now from Equation 3.1 and assuming δ as threshold, we have

$$\begin{aligned} \frac{X-Y}{X} &= \delta \\ X - Y &= X * \delta \\ \Rightarrow \qquad Y &= X * (1-\delta) \end{aligned} \tag{3.2}$$

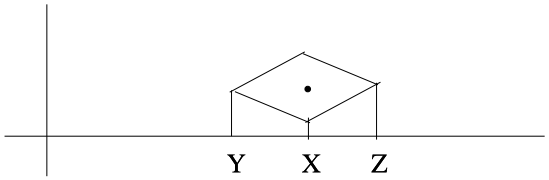


Figure 3.2: Variable-Sized Cluster

and

$$\begin{aligned} \frac{Z-X}{Z} &= \delta \\ Z-X &= Z*\delta \\ \Rightarrow \quad Z &= X/(1-\delta) \end{aligned} \tag{3.3}$$

The Equation 3.2 and Equation 3.3 can be used to determine value of threshold (δ) from cluster budget i.e., we have to find appropriate value of threshold such that given number of clusters can cover plan space.

Example 2 Consider a query template[4] in which K of its relations have selection predicate with bind variables. Clearly the plan space corresponding to this query template is K dimensional with each dimension corresponding to selectivity of one relation. Suppose we have budget of $2N$ clusters to cover this plan space. Since the shape of cluster is like rhombus, to cover the plan space, we will need two K dimensional grids of cluster representatives with each grid having total of N representatives and $\sqrt[k]{N}$ representatives along each dimension. □

With the help of Example 2, We will now illustrate use of above equations to determine position of the representatives and appropriate value of threshold. Let threshold be δ. First, we will determine the position of last representatives along any of these dimensions. The cluster corresponding to the last representative should have stretch up to selectivity of 1. This will be achieved if we keep the cluster representative at selectivity $(1-\delta)$ (Equation 3.3) and the cluster will have stretch from $(1-\delta)^2$ to 1 (Equation 3.2). By similar argument, second last cluster should have stretch upto $(1-\delta)^2$ and its representative should be placed at $(1-\delta)^3$. Inducting the above argument, i^{th} last representative should

[4] A query template represents a query in which some or all of the constants in the where-clause predicates have been replaced by bind variables.

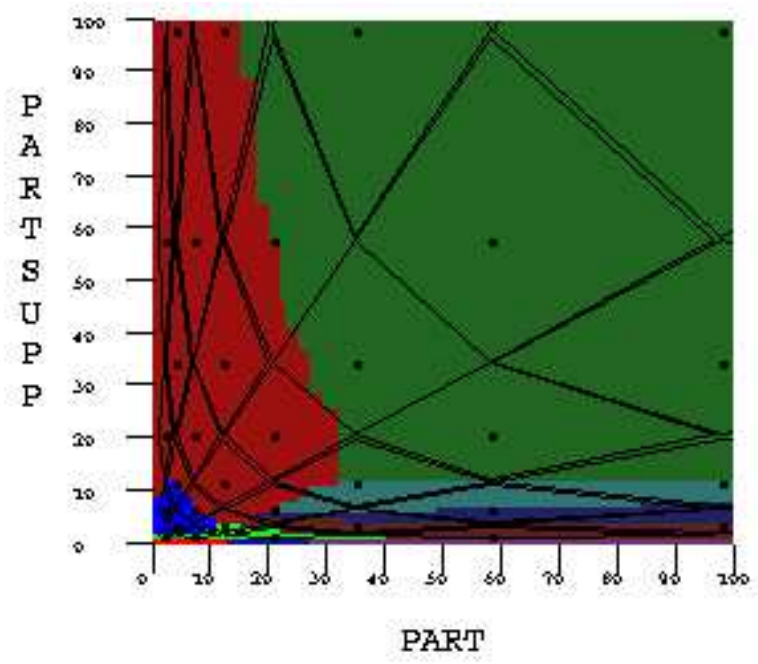


Figure 3.3: Variable-Sized clustering

be placed at $(1-\delta)^{2*i-1}$ and the first representative ($\sqrt[k]{N}^{th}$ last representative) should be placed at $(1-\delta)^{2*\sqrt[k]{N}-1}$. Assigning this position to some desired value (say 0.01), we can determine δ. From this value of δ, we can also determine positions of representative to cover the plan space.

A sample output of variable-sized clustering for a SPJ version of Query 2 from the TPC-H benchmark [23] is shown in Figure 3.3. From figure, we can see that variable-sized clustering has covered the plan space in much better way as compared to fixed-sized clustering (Figure 3.1).

3.3 Decision-Tree Classifier

We now move on to the problem of efficiently determining which cluster, if any, a new query should belong to. There are a host of classification schemes that can be applied to the query classification problem.

PLASTIC Version-1 system uses the Leader algorithm itself for classification purposes, and in fact, make it an *online* classifier by having the optimizer generate a plan whenever a new representative is encountered. In the Version-1 system, identifying the matching cluster (if any) for the new query, was achieved by comparing the new query with the cluster representatives until either a similar representative

was found, or all were found to be dis-similar. Since, matching with each representative would require getting global/table features for the representative, finding optimal mapping between tables and then computing distance using distance function, the matching process can become computationally expensive when a large number of clusters are present, as would often be the case.

Accordingly, we need a faster technique such as decision tree or hierarchical clustering [3] – we have explored the former option since it naturally suits our problem. This is because most of our query features are deterministic as well as common to a small group of clusters and we can therefore have a set of comparisons that zero-in on the required cluster very quickly. For example, the feature set: Degree Sequence and Join Predicate Index Counts, will be the same for all the queries within the cluster and thus can be considered to be a characteristic of the cluster acting as a decision rule for selecting clusters. Another important advantage of decision trees is that once we have the rules generated, we can even drop the source query feature vectors and simply interpret clusters as leaves of the decision tree.

We have now incorporated a new classifier module into the PLASTIC architecture which operates on the clusters in the database, after grouping them based on plan commonality – that is, clusters sharing the same plan are grouped together and a classifier is built on these groups. Specifically, the popular C 5.0 [25] decision-tree classifier has been integrated into our prototype.

To optimize the grouping process, initially the plan template of each query representative is traversed in post-order and an MD5 hash signature of this traversal is computed. Subsequently, these signatures, rather than the plan templates themselves, are compared to decide plan commonality among clusters. Such grouping significantly reduces the number of class labels in the cluster database, and has twofold advantages: Firstly, it increases the accuracy of the classifier, and secondly, results in a decision tree of lesser height, thereby requiring lesser time for classification. Note that, grouping of clusters do not affect the accuracy since all clusters which were grouped together have the same plan templates. Quantitatively, our experiments show that the cluster identification time reduces by an *order of magnitude*, at only a small cost in the overall matching accuracy (Section 5.2). In fact, with the classifier, the identification cost is proportional to the *heterogeneity* of the clusters, whereas in the earlier version, the cost was proportional to the *cardinality* of the clusters.

Apart from improving the speed of cluster identification, the decision tree also helps to identify the attributes of the query feature vector that have the most impact on plan choices. We expect that this information will be especially beneficial for both database administrators and query optimization researchers/students. A sample output of the classifier module is shown in Figure 3.4.

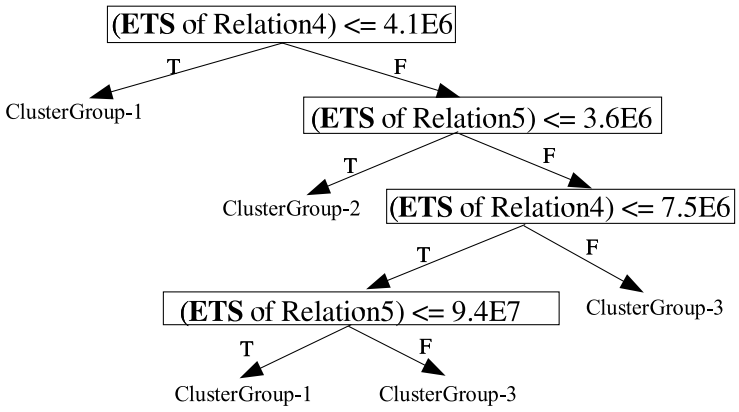


Figure 3.4: Classifier Module Output

3.4 Including Cost Estimations in Generated Plans

The plans generated by PLASTIC Version-1 system were incomplete in that they did not include the associated *operator costs*, although this is a standard feature of plans generated by the optimizer. Note that the costs of the cluster representative's plan cannot be directly used for this purpose since there can be significant variation in the matched queries, although they share a common plan template. To address this issue, we have included an estimation module that scales and interpolates the costs of the cluster representative to reflect the costs of the new query, and includes this information in the generated plan.

Since the new query and its matched cluster representative may differ on selectivities of multiple relations, a *multivariate* strategy is required to compute the interpolated values. Currently, we have implemented first order multivariate interpolation based on barycentric coordinates [26], which has a low time complexity and require to access representatives from database only one more than number of relations on which query and matched representative differs.

Our experiments on various TPCH queries shows that, this module predicts the cost with 96.33 percent accuracy on the average case and nearly 80 percent in worst case. The reason for low worst case percentage accuracy is because of low cost for highly selective query and small absolute error in prediction results in high percentage error.

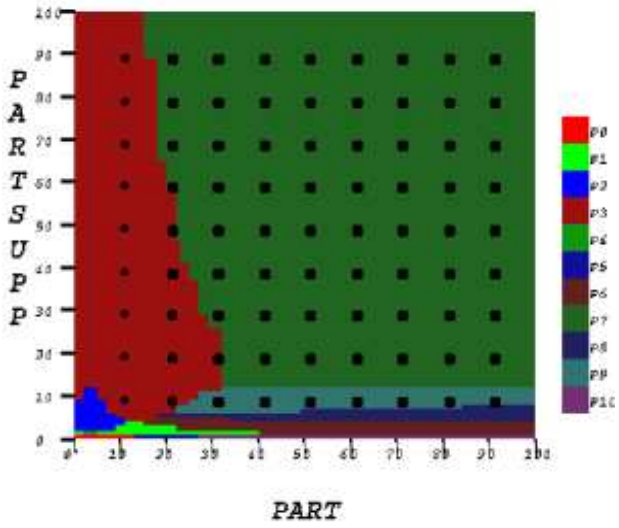


Figure 3.5: Plan-Cost Diagram

3.5 Automated Plan-Cost Diagram Generator

Generating a "plan diagram"[5] is a computationally expensive process since it involves firing of a large number of queries to get optimizer plan choices and comparing among these plan choices. In PLASTIC Version-1 system this diagram was constructed from scratch for each time.

We have now added a facility for persistently storing these diagrams directly in the database, from which they can be immediately retrieved at the desired time. We have also mapped this stored plan diagrams with clusters in PLASTIC, to aid in visual interfaces for generating, viewing, and reorganizing query clusters. Further, we have augmented the plan diagram, which is a qualitative picture, to include quantitative costs, thereby resulting in a combined plan-cost diagram. An example plan-cost diagram[5] corresponding to an SPJ version of Query 2 from the TPC-H benchmark [23] is shown in Figure 3.5. Note that there are eleven plan optimality regions (P0 through P10) spanning the entire selectivity spectrum on the PART and PARTSUPP relations, and the sizes of the dots indicate the relative plan costs.

[5]The diagram is based on plan selections obtained with the IBM DB2 query optimizer.

3.6 Plan Analyzer Module

The PLASTIC Version-1 system was having facility to view "plan diagram"[5], which is an enumeration of the plan choices of the optimizer over the query space. Based on complexity of plans, it may be difficult to find out differences between different plan choices. So, a new Plan Analyzer module, intended for analyzing the specific differences between a pair of query execution plans, has been incorporated in the Plastic tool. We expect that this module will be especially beneficial for database administrators and query optimization researchers/students to help understand plan choices made by the optimizer. It can be used in two ways: (a) to compare plan choices for different versions of a query on a single platform, or (b) to compare choices for the same query across database platforms.

3.6.1 Change Detection Algorithms

Mots of work in change detection has focused on computing differences between flat files e.g., GNU *diff* utility. This program uses the LCS (Longest Common Subsequence) algorithm [10]. In [1], it was pointed out that this techniques can not generalized to handle structured data. Since, the plans can be viewed as tree, comparison between plans can be treated as comparison between trees or hierarchically structured information and it is natural idea to utilize tree-to-tree correction techniques [1, 7, 19, 20, 22]. The tree-to-tree correction problem is to find best (lowest cost) edit script[6] to convert one tree into another tree. All above algorithms except [19] uses basic edit operation like insert leaf, delete leaf, change node label and optionally move the node. These algorithms works in two phases – in first phase, they find best maximal matching between two trees and in second phase they use this matching to produce best editing script. For our case, we are interested in first phase only.

We have adapted the X-Diff [20] based comparison algorithm, originally intended for XML documents, as it best suits our needs. The algorithm proposed in [1] does not guaranteed to give optimal results [20]. Pattern matching problem considered in [7] is different from tree matching problem – authors have considered the problem of finding occurrences of subtree or subtree pattern in some other tree. A polynomial-time algorithm proposed in [22] is based on restriction that matching can be only performed between nodes at same level which is clearly not be acceptable in plan comparison.

[6]Edit script is sequence of edit operations.

```
INPUT    : Plan Tree T1, T2
OUTPUT: matching between nodes of T1, T2

GetMatching (Tree T1, Tree T2)
{
   // initialize matching
1. matched-nodes = φ

   // get leaves and join nodes
2. leaves1 = all leaves in Tree T1,      leaves2 = all leaves in Tree T2
   join1 = all join-nodes in Tree T1,    join2 = all join-nodes in Tree T2

   // add matching leaves to matched-nodes
3. FOR each l1 in leaves1 and l2 in leaves2 such that l1, l2 represent
   same table/index
        add (l1, l2) into matched-nodes

   // add matching join-nodes to matched-nodes
4. FOR each j1 in join1 and j2 in join2 such that j1, j2 represent same
   join method and inner,outer relations of j1, j2 are same
        add (j1, j2) into matched-nodes

   // recursively add parents of matched nodes
5. FOR each matched-pair (n1, n2) in matched-nodes
      f1 = parent of n1, f2 = parent of n2
      WHILE (f1, f2 are not root and f1, f2 are not join-nodes and
      f1, f2 represent same operation)
         add (f1, f2) into matched-nodes
         f1 = parent of f1, f2 = parent of f2

6. RETURN (matching);
}
```

Figure 3.6: The Matching Algorithm

3.6.2 The Matching Algorithm

The matching algorithm returns the best maximal one to one mapping between nodes of two plan-trees. Note that this mapping do not necessarily involve all nodes from both trees. Some of the nodes in both tree may not be mapped to any node in other tree.

Since the X-Diff matching algorithm uses the notion of 'node signature', which is based on XML structure, we have modified original matching algorithm. Our matching algorithm is based on following

observations.

1. A leaf node represents table/index and it can be matched only to leaf node in other tree.
2. Leaf nodes are matched only if they represent same table/index.
3. Node representing join operation can be matched only to a node representing join operation in other tree.
4. Nodes representing join operation are matched only if they use same join method and inner and outer relations to be joined are same.
5. Internal nodes with only one child node represent the operation on single data source.
6. Internal nodes having one child matches if their children are matched and they represent same operation e.g., a *TBSCAN* operation on table *NATION* can match with only *TBSCAN* operation on table *NATION* in other tree.

The pseudo-code of our matching algorithm is given in Figure 3.6. We first collect leaves and internal nodes representing join operation. Each leave from first tree is checked for matching in the leaves of other tree. This matching can be optimized if we sort the leaves based on table/index names they represent. The matching in sorted leaves can be found by a sequential scan of both leaves as in case of *MERGE-SORT JOIN*. Further table and index leaves can be separated and checked for matching separately. Once matching between leaves is established, we establish matching between nodes representing joins. Matching between join nodes can be optimized by grouping the nodes based on join method. Finally, for each pair of matched nodes, we recursively add their parent if they represent the same operation.

Lets assume that there are r relations in query for whom plan choices are compared. The step 3 of matching algorithm will have major cost in sorting leafs and would have $O(r * \log r)$ time complexity. The join nodes in plan-tree will be one less than the number of relations ($r - 1$ in this case). Step 4 of algorithm is optimized by grouping nodes based on join method but it may not help in worst case when all join operations in plan are using same method. So step 4 of algorithm, which is most expensive, will have worst case time complexity of $O(r!)$. Since, no operation like sorting, filtering which operates on single data source is repeated on same data source, step 5 of algorithm will have time complexity of $O(r)$. Thus overall worst case time complexity of matching algorithm is $O(r!)$.

A sample output of this module on Plan 7 and Plan 9 of Figure 3.5 is shown in Figure 3.7. From this figure, we can see that these two plans differ only in the access method (*index scan* versus *table scan*) for

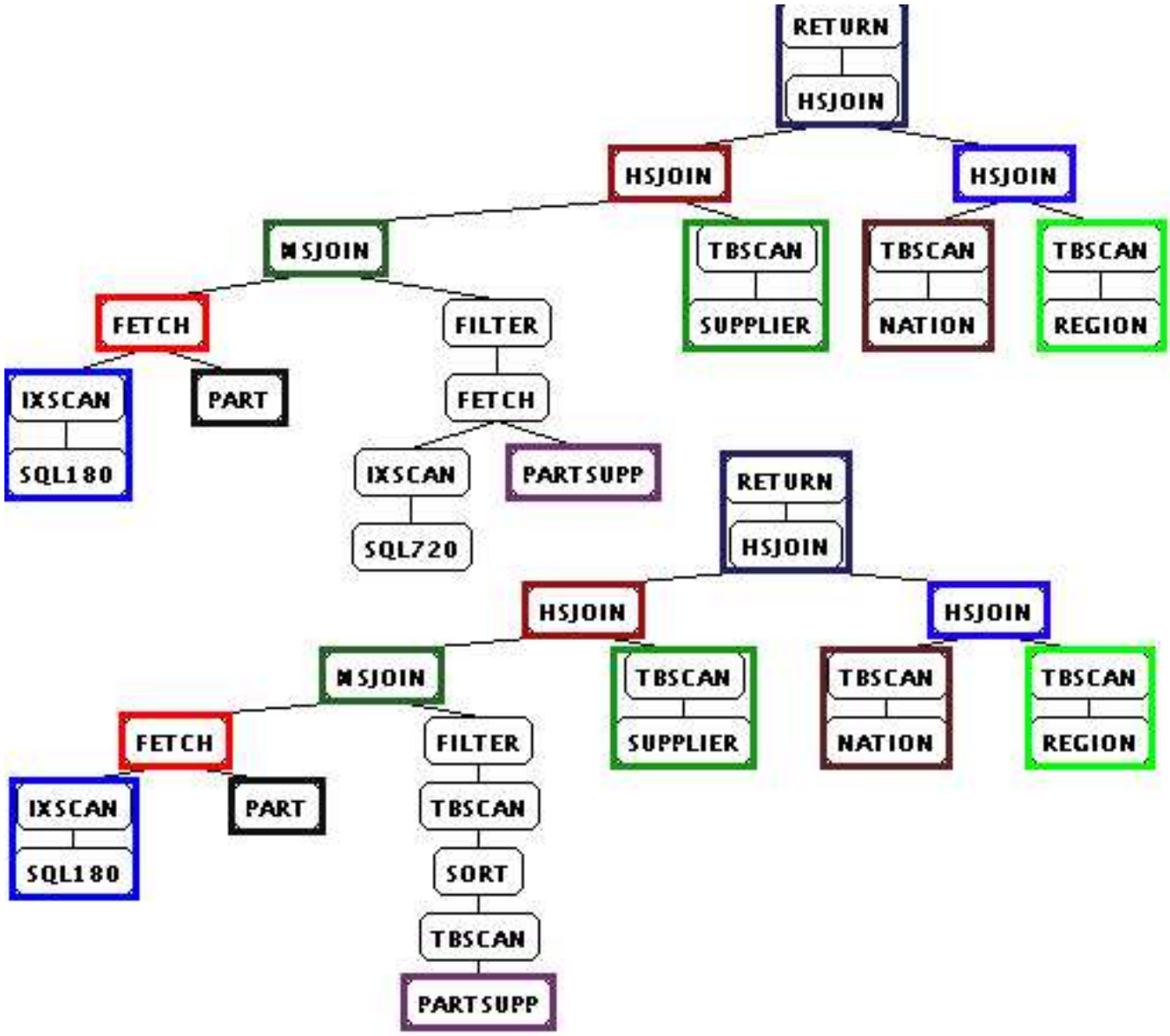


Figure 3.7: Plan Difference

the PARTSUPP relation. In Plan 7, sorting is not required on *PARTSUPP* relation because it is accessed through index which will return tuples in sorted order itself.

Chapter 4

Implementation

PLASTIC tool is implemented in Java and is hosted on two commercial database platforms namely, DB2 and Oracle. It is implemented as three tier architecture as shown in Figure 4.1. The topmost tier implements GUI which is responsible for getting user requests and displaying results back. Second tier implements the complete PLASTIC logic which includes clustering mechanism. This tier also includes packages to communicate with specific database system. The database system, which is backbone of this tool, forms the third tier.

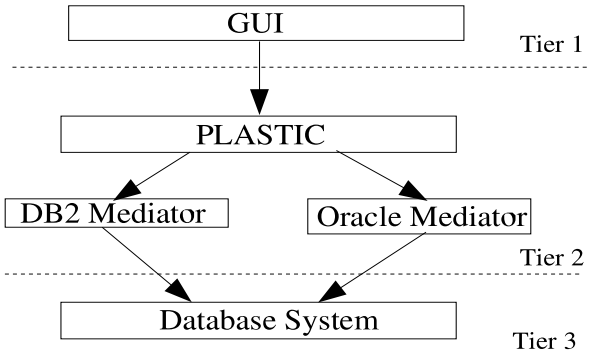


Figure 4.1: PLASTIC Implementation's Architecture

The implementation is specifically oriented toward demonstrating how PLASTIC can help to (a) significantly amortize the overheads of query optimization in database systems, and (b) serve as a research, educational and administrative tool for understanding the intricacies of query plan generation.

Chapter 5

Performance Framework

We conducted a variety of experiments to evaluate the *accuracy* and *efficiency* of the PLASTIC algorithm with regard to cluster identification. In this chapter, we describe the experimental framework and the results.

5.1 Experimental setup

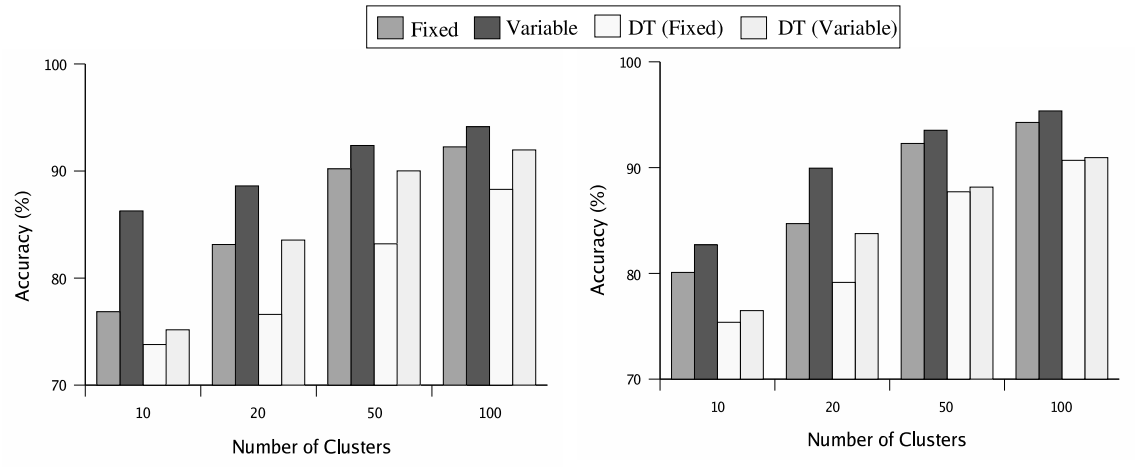
PLASTIC has been evaluated on the TPC-H benchmark database populated at scale 1 (i.e. 1 GB of data). The tables present in the database and their sizes are given in Table 5.1. The database was populated using DBGEN [23] and the queries were generated using QGEN [23] – these queries were modified to result in un-correlated SPJ queries.

All our experiments were conducted by running PLASTIC on RedHat-8 based Linux workstation having 1GB RAM, 40 GB disk and Pentium-4 2.4GHz processor. We have evaluated PLASTIC on both DB2 and Oracle optimizer. Since we hope that PLASTIC tool will allow optimizer to run at highest optimization level, both of these optimizers were running at highest optimization level. We have also enabled hash join in DB2.

Table	Cardinality	Size (in MB)
REGION	5	$4 * 10^{-4}$
NATION	25	$2.1 * 10^{-3}$
SUPPLIER	10000	2
CUSTOMER	150000	26

Table	Cardinality	Size (in MB)
PART	200000	30
PARTSUPP	800000	110
ORDERS	1500000	149
LINEITEM	6000015	641

Table 5.1: TPC-H Table Statistics



(a) DB2 **(b) ORACLE**

Figure 5.1: Accuracy of PLASTIC

We have considered and compared four clustering mechanisms – fixed-sized clustering, variable-sized clustering and decision tree on fixed-sized clusters and decision tree on variable-sized clusters. These clustering mechanisms are compared with *accuracy* and *efficiency* as metrics for evaluation. The accuracy was computed as for how many of fired queries plan predicted by PLASTIC matched the plan given by optimizer. We measure efficiency in terms of the time taken for classification. That is, given K clusters how much time does it take PLASTIC to classify a new query and predict a plan.

5.2 Experimental Results

Our experiments evaluates the performance for a query workload that is generated from the same query template. The numbers reported here are average over five query templates from tpch-based queries.

Our result are based on firing the several hundreds of queries uniformly distributed within query-space. We have also varied the number of clusters present in system per query template and number of cluster on which decision tree is build. Threshold was varied based on cluster budget so that complete query space is covered.

5.2.1 Accuracy

We begin by the accuracy of PLASTIC. The associated performance perspective is shown in Figure 5.1, which presents accuracy of PLASTIC with different clustering mechanisms and for different numbers of

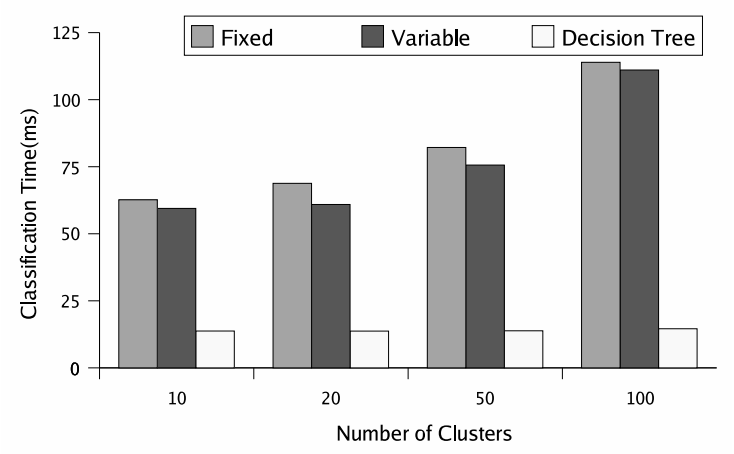


Figure 5.2: Efficiency of PLASTIC

clusters per query template. As can be seen in the figure, with as less as ten clusters per query template, PLASTIC is able to deliver the accuracy more than 80 percent. Moreover, independent of specifics of clustering mechanism used, as number of clusters to cover plan space increases, accuracy of PLASTIC also increases. This is expected because with more cluster, threshold and cluster size decreases and stretch of cluster that straddle across plan boundaries is less. Importantly, PLASTIC can deliver an accuracy of above 90 percent, which shows that PLASTIC can be profitably used in conjunction with a traditional optimizer.

Another point to note is that variable-sized clustering have increased the accuracy of PLASTIC significantly, particularly for small number of clusters. We have even found out, in some cases, variable-sized clustering reduces error by as much as 50 percent. Lastly, use of decision tree, a faster clustering mechanism, decreases the accuracy of PLASTIC. The effect is mostly visible for small number of clusters. With more clusters, effect of faster clustering mechanism also decreases. So we propose to build decision tree only on sufficient number of clusters.

5.2.2 Efficiency

Having established high accuracy of PLASTIC, we now move on to evaluating it's efficiency. In Figure 5.2, we show classification timing in milliseconds on DB2, for different clustering mechanism and

for different number of clusters. Classification timing on Oracle are similar[1]. From the figure, it is clear that, because of decision tree optimization, cluster identification time reduces always by an *order of magnitude*. The cost of decision tree classification is almost constant because it is depends upon height of tree and not on number of clusters on which it was built. In fact, with the classifier, the identification cost is proportional to the *heterogeneity* of the clusters (plan space), whereas in the other cases, the cost was proportional to the *cardinality* of the clusters. As pointed out earlier, this benefit is at small cost in the identification accuracy.

A deterrent to always run the optimizer at level 9 is the fact that this level involves considerable additional computation cost. However, with PLASTIC, we can now *afford* to incur this cost since it is a one-time cost only. Hence, the 90% of cases where PLASTIC is correct would run at the *best* optimization level, which would have been difficult to achieve otherwise. In summary, this experiment clearly shows that *very substantial improvements in the query optimization process can be realized through a properly designed query clustering technique*.

[1]Classification is independent of database systems as during classification database is used only to retrieve representative's feature vectors.

Chapter 6

Conclusions

We have presented a host of new features into PLASTIC to extend its scope, useability, and efficiency.

We have proposed modifications into feature vector to increase the query matching capabilities of PLASTIC. We have proposed variable-sized clustering to address the twin problem of insufficient clusters in the high-volatility regions of the plan space and redundant clusters in the low-volatility regions. It was based on intuition that normally high-volatility region is present at highly selective region of plan space. We have also shown how threshold value can be determined from clusters budget. Through experiments on TPC-H benchmark database, we have shown that using variable-sized clustering classification errors can be reduced significantly.

We have incorporated C5.0 decision tree into PLASTIC for fast matching cluster identification. We group the cluster based on plan commonality first and then classifier was build on these group. Our experiments on TPC-H benchmark database have shown that classification time was reduced by an orders of magnitude.

We have also added plan analyser module, which can be used to compare plan choices for different version of same query or to compare plan choices across the database platforms. We have added cost estimation module into PLASTIC to estimate the cost of plan for new query from plan templates stored at cluster representatives. Further, we have augmented the plan diagram, which is a qualitative picture, to include quantitative costs, thereby resulting in a combined plan-cost diagram.

Bibliography

[1] S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom, "Change detection in hierarchically structured information", *Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 493–504*,June 1996.

[2] R. Cole and G. Graefe, "Optimization of Dynamic Query Evaluation Plans", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[3] R. Duda and P. Hart, "Pattern Recognition and Scene Analysis", *John Wiley, New York*, 1973.

[4] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms", *Proc. of 24th Intl. Conf. on Very Large Data Bases (VLDB)*, August 1998.

[5] 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.

[6] R. Gopal and R. Ramesh, "The Query Clustering Problem: A Set Partitioning Approach", *IEEE Trans. on Knowledge and Data Engineering*, 7(6), December 1995.

[7] C. Hoffman, M. O'Donell, "Pattern matching in trees", *Journal of the ACM*, 1982.

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

[9] Y. Ioannidis, R. Ng, K. Shim and T. Sellis, "Parametric Query Processing", *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, 1992.

[10] E. W. Myers, "An O(ND) Difference Algorithms and Its Variations", *Algorithmica*, 1(2): 251-266, 1986.

[11] J. Park and A. Segev, "Using common sub-expressions to optimize multiple queries", *Proc. of IEEE Intl. Conf. on Data Engineering (ICDE)*, 1993.

[12] P. Roy, S. Seshadri, S. Sudarshan and S. Bhobe, "Efficient and Extensible Algorithms for Multi Query Optimization", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[13] 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.

[14] T. Sellis, "Multiple Query Optimization", *ACM Trans. on Database Systems*, 13(1), March 1988.

[15] V. Sengar and J. Haritsa, "PLASTIC: Reducing Query Optimization Overheads through Plan Recycling", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[16] V. Sengar, "PLASTIC: Reducing Query Optimization Overheads through Plan Recycling", *Master's Thesis*, CSA, Indian Institute of Science, March 2003.

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

[18] M. Stonebraker, J. Frew, K. Gardels and J. Meredith, "The SEQUOIA 2000 Storage Benchmark", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[19] K. Tai, "The tree-to-tree correction problem", *Journal of the ACM*, 26(3):422–433, July 1979.

[20] Y. Wang, D. DeWitt and J.Cai, "X-Diff: A Fast Change Detection Algorithm for XML Documents", *Proc. of the 19th IEEE Intl. Conference on Data Engineering*, March 2003.

[21] T. Zhang, R. Ramakrishnan and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[22] K. Zhang, "A New Editing based Distance between Unordered Labeled Trees", *Proc. CPM Combinatorial Pattern Matching*, LNCS 684, 254-265, 1993.

[23] http://www.tpc.org/tpch/

[24] http://download-east.oracle.com/otndoc/oracle9i/901_doc/server.901/a87503/toc.htm

[25] http://www.rulequest.com/see5-info.html

[26] www.library.uu.nl/digiarchief/dip/diss/2003-1028-125323/appd.pdf