Plan Selection based on Query Clustering

Antara Ghosh Jignashu Parikh Vibhuti S. Sengar Jayant R. Haritsa*

Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, INDIA
{antara,jignashu,vibhuti,haritsa}@csa.iisc.ernet.in

Abstract

Query optimization is a computationally intensive process, especially for complex queries. We present here a tool, called PLASTIC, that can be used by query optimizers to amortize the optimization cost. Our scheme 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 a commercial optimizer show that PLASTIC predicts the correct plan choice in most cases, thereby providing significantly improved query optimization times. Further, when errors are made, the additional execution cost incurred due to the sub-optimal plan choices is marginal.

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 for the query [16]. 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 [1].

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. In this paper, we present a value-addition tool for query optimizers that *amortizes* the cost of query optimization through the *reuse* of plans generated for earlier queries. Specifically, the tool stores a database of plans[1] and attempts to assign one of these plans to the new query with the expectation that the selected plan would be the same as that generated by the optimizer; only if no suitable assignment can be found is the optimization process actually carried out and the newly-generated plan is added to the *plan database* for future use.

Our tool, called **PLASTIC** (PLAn Selection Through Incremental Clustering), has been developed for the relational database framework and is based on the following approach: First, we define a query *feature vector* comprised of information that can be determined from the query and from the catalogs of the RDBMS. This information includes both overall structural features such as the number of tables and joins in the query, as well as table-specific features such as the presence of indexes on query attributes, the number of predicates in which the table is involved, and the size of the table. Next, we define a similarity function that takes a pair of query feature vectors as input and quantitatively computes their separation in feature space. A threshold value for this separation is used to decide whether or not the two queries are similar.

Then, using this similarity definition, *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. In our current implementation of PLASTIC, a *leader* algorithm [9] is used to determine cluster representatives, and a *decision-tree* [12] is constructed for classification purposes.

A critical feature of our definition of similarity among queries is that it is determined, as mentioned above, based on their feature vectors. An alternative approach could have been to first submit a large set of queries to an optimizer and then group those queries for which the same plan is generated by the optimizer. However, the problem with this approach is that there may be queries which are

*Contact Author

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

[1] As explained later, it is plan *templates* that are actually stored.

very different in feature space but the optimizer may generate the same plan for them. This means that classifying a new query to a cluster may entail the expensive process of comparing against *all* the queries present in the cluster.

With our approach, however, since all queries in a cluster are similar in feature space, a *single* query can become the representative of the cluster, and therefore only one distance computation is required to check whether a new query belongs to the cluster. In short, we cluster queries in the feature space such that all the queries in a cluster have matching plans in the plan space. This reduces the complexity of classification from $O(N)$ to $O(K)$, where N is the number of queries and K is the number of clusters. A fallout of our approach is that more than one cluster may map to the *same* execution plan. That is, there is an $N:1$ relationship from the cluster space to the plan space. While this may appear redundant at first sight, the advantage is that it facilitates more accurate assignments of queries to clusters. Further, the redundancy can be easily minimized, as explained later in the paper.

The efficacy of our tool can be evaluated with respect to a variety of metrics: Firstly, its *accuracy* in terms of correctly anticipating the plan choice that would have been made by the optimizer for the same query. Secondly, its *efficiency* in terms of the time taken to *select* a plan as compared to the time taken by the optimizer to *generate* the plan. Thirdly, its *error penalty* in terms of the potential increase in query execution time that is incurred in those cases where the tool makes a plan choice *different* from that of the optimizer. Lastly, its *space overhead* in terms of the size of the plan cache.

In terms of the above performance framework, we would ideally like the tool to have high accuracy, high efficiency, low error penalty, and low space overhead. To evaluate this quantitatively, we have conducted a detailed performance study of PLASTIC with regard to the DB2 (Universal Database Version 7.0) optimizer against a variety of queries including a representative set from the TPC-H benchmark. We make two simplifying assumptions in our study: Firstly, we assume that the system state is constant and therefore resource-related issues such as buffer allocation or disk clustering are not considered. Secondly, we restrict our attention to pure SPJ queries. Under these conditions, our experimental results indicate that PLASTIC achieves over 90 percent accuracy, an order of magnitude improvement in efficiency, an error penalty of less than 10 percent, and a low space overhead.

To the best of our knowledge, the work described here represents the first research initiative on clustering-based plan selection.

2 Problem Motivation and Framework

We wish to come up with a system for clustering queries efficiently in a manner such that, with an acceptably high probability, the query execution plans chosen by our system match the plans that would be chosen by the optimizer for the same queries. To motivate this objective, consider

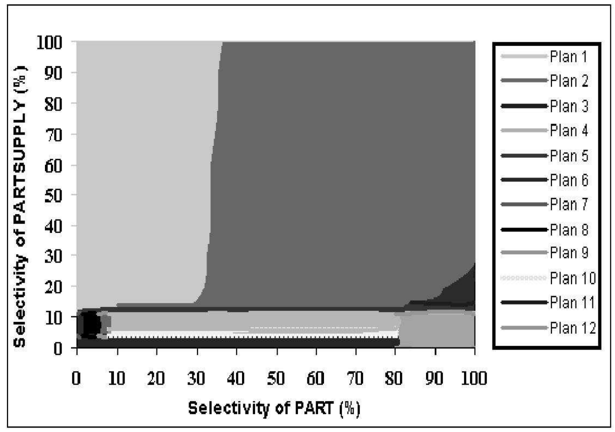

Figure 1: **Plan Diagram for Q2'**

query template Q2' shown below, which is a simplified version of query Q2 from the TPC-H benchmark (the nested sub-query and group-by operations have been removed).

Query 2':

```
select
   s_acctbal, s_name, n_name, p_partkey,
   p_mfgr, s_address, s_phone, s_comment
from
   part p, supplier s, partsupp ps,
   nation n, region r
where
   p_partkey = ps_partkey and
   s_suppkey = ps_suppkey and
   p_size = :1 and p_type like :2 and
   s_nationkey = n_nationkey and
   n_regionkey = r_regionkey and
   r_name = :3 and ps_supplycost = :4
```

Here, the `:1` through `:4` are "bind variables" that are replaced by constants in an actual query. For this query template, we show in Figure 1 a sample *plan diagram* obtained on the DB2 optimizer by varying the selectivities of tables PARTSUPP and PART in the query, keeping the rest of the query parameters constant. The plan diagram shows which plan is selected by DB2 for each pairwise combination of selection predicate selectivities. Note that there are a total of 12 different plans, with Plan 1 and Plan 2 covering the majority of the space.

Consider Plan 1 – if we know apriori that a newly-arrived query can be mapped to this region, we can assign Plan 1 to the new query. To realize this practically, PLASTIC divides the query space into clusters, with every cluster represented by a particular query that belongs to the cluster. The clustering technique ensures that the same plan assignment holds for all queries falling into the space represented by a particular cluster. This is shown pictorially in Figure 2, where the clusters are the semi-elliptical structures (the explanation for the cluster shapes is given later in Section 5).

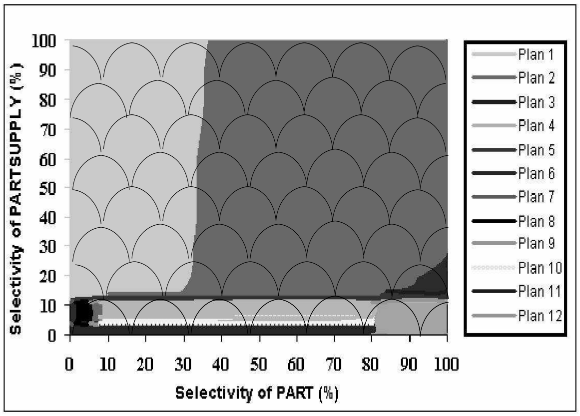

Figure 2: **Clusters in the Plan Diagram for Q2'**

A variety of design issues need to be addressed to efficiently achieve the desired clustering. These issues can be briefly stated as: (a) *query representation*: what are the features used to represent a query? (b) *query similarity*: when can two queries be called similar? (c) *query clustering*: what mechanism is to be used for clustering similar queries? (d) *cluster size*: what should be the size of each cluster? and (e) *query classification*: how are new queries classified into the given set of clusters?

We address these issues in the remainder of this paper. For ease of exposition, we first define the following basic set of concepts:

Query Template: A query template represents a query in which some or all of the constants 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.

Query Template Space: This is the set of different queries that can be instantiated from a query template by assigning different values to the bind variables in the query template.

Plan Diagram for a Query Template: The Plan Diagram for a query template is an enumeration of the plans chosen by the optimizer over all points in the associated query template space. The number of dimensions of the plan diagram is equal to the number of tables of the query template that have selection predicates on them. Figure 1 is an example of a two-dimensional plan diagram with bind variables corresponding to the PART and PARTSUPP tables.

Plan Template: A plan template is a query plan wherein all the database operators (e.g. TABLESCAN, SORT, MERGE-JOIN) are retained but the specific values of the inputs to these operators such as the table and attribute names have been replaced by variables.

Two query plans are said to match if their plan templates are the same.

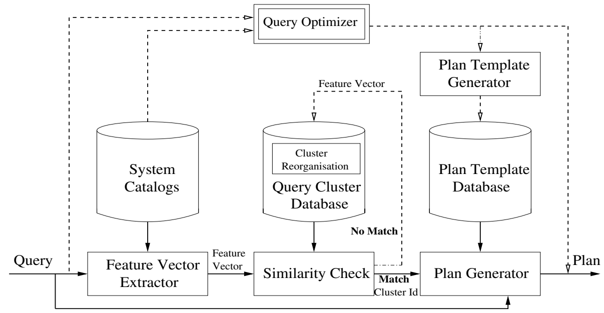

Figure 3: The PLASTIC Architecture

Query Cluster and Radius: A cluster of queries is defined as a set of queries that are similar as per a predefined similarity metric. The radius of the cluster is the maximum distance of any element from the center of the cluster. While the center could be defined in a variety of ways – for example, as the centroid of the cluster, here we use the position of the cluster representative to represent the center.

3 System Overview

We now present an overview of the PLASTIC architecture, whose block-level diagram is shown in Figure 3. 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 given to the system is first parsed by 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 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.

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 the specifics of the input query.

On the other hand, if no matching cluster is found (dashed lines in Figure 3), 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.

The detailed functioning of the modules described above are discussed in the following sections.

4 Query Representation

We start off by addressing the issue of query representation since the appropriate choice of features forms the core of any clustering approach. 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 DB2 optimizer over a broad spectrum of queries.

4.1 Structural Features

We will use Figure 4 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 the number of *join* predicates in which a particular table is involved. For example, the degree of table E in Figure 4(a) is 3. 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.

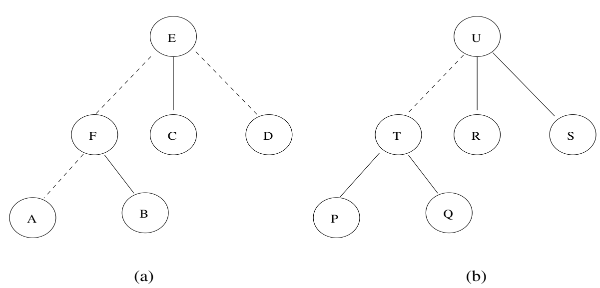

Figure 4: Degree Graphs

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 4 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 [15], 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 4, 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.

4.2 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 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.

Putting all of the above together, our complete query feature vector definition is as shown in Table 1. For ease of understanding, we have separated the features into *Global*

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 1: Query Feature Vector

Features, which are query-wide values, and *Table Features*, which are relevant to individual tables.

4.2.1 Example Feature Vector

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. 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. Note also that the ETS values for A and B are the same as their TS values because there are no *selection* predicates on either table. We 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.

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

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	200000	100000

Table 2: Example Query Feature Vector

5 Establishing Similarity

We now move on to the next issue of determining when two queries are said to be similar. 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 = 'Brand#15'
```

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.

5.1 The SIMCHECK Algorithm

The SIMCHECK algorithm, whose pseudocode is shown in Figure 5, 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 dissimilar 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.

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 5: **The SIMCHECK Algorithm**

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 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.

5.1.1 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 4(a) is (2, 1) since there are two SARGable predicates and one non-SARGable predicate. Similarly, for table U in Figure 4(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.

5.1.2 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 4.2, 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.

Note that $dist_{ij} \leq 1$ and thus $dist$, the sum of the distances of all the table pairs for a particular mapping, is bounded above by the total number of tables in the query. When there are multiple mappings possible between the tables of two queries, the mapping with the minimum value of $dist$ is chosen. For queries to be considered similar, $dist$ should be less than a particular user-defined *Threshold*.

The $dist_{ij}$ function essentially computes the separation between two tables in terms of their TS and ETS values with respect to the query. The reason that we are considering not only the cardinalities of the join input tables but also their tuple sizes is because the choice of query plans depends on the *total* amount of data that has to be fetched and stored in the buffers – that is, the length of the projected attributes has a direct impact on plan choice even if all other parameters are the same. The numerator is normalized by the maximum of the sizes of the two relations in order to keep the value of the function bounded between 0 and 1. Another feature of the distance function is that it is symmetrical, as can be readily seen from the formula.

Typically, w_1 is set higher than w_2. This is because the impact of the table size on the plan choice is usually much more than that of the effective table size. The threshold is a parameter that decides how "tight" we want the similarity to be, that is, it determines the cluster radius. The smaller this value, the lesser the chance of errors, but the more expensive it becomes to do query classification due to the increased number of clusters. The weights, w_1 and w_2, and the Threshold are currently evaluated empirically, but in our future work we plan to investigate automated mechanisms for setting these parameters.

5.1.3 Complexity of Mapping

Mapping tables between two queries can turn out to be an expensive task. This is because the complexity of any such algorithm is $O(n!)$ where n is the number of tables involved in the query. SIMCHECK is optimized for reducing this time complexity by grouping the tables into sets of tables having the same degree. Specifically, if Q1 and Q2 have N tables that are divided by a partition $p_1, p_2, \ldots, p_k$ based on their degrees, then we reduce the problem from an N:N scenario to matching individual groups. Thus, the complexity of the algorithm now becomes $O(p_1!+p_2!+p_3!+...+p_k!)$ which in the average case is far less than O(n!). For example, for the query represented in Figure 4 (a), the degree-based partitions would be $\{\{E\},\{F\},\{A,B,C,D\}\}$ and the cost of mapping it to the query in Figure 4 (b) with partition $\{\{U\},\{T\},\{P,Q,R,S\}\}$ is 4!+1!+1!, which is considerably less than the 6! cost of a global mapping strategy. Further, the fact that only compatible table pairs are chosen for mapping further reduces the amount of computation.

5.2 Effectiveness of the Algorithm

We now present a few sample scenarios that highlight the effectiveness of our algorithm. Consider the following set of simple queries, QA, QA' and QA", operating on the TPC-H tables NATION and REGION:

QA:

```
select * from nation, region
where n_nationkey=r_regionkey
```

QA':

```
select n_nationkey from nation, region
where n_nationkey=r_regionkey
```

QA":

```
select n_comment, r_comment
from nation, region
```

The QA and QA' queries appear very similar syntactically but when given to the DB2 optimizer, they produce rather different execution plans. For QA the plan chosen by DB2 does an index-based table-fetch through tables NATION and REGION while for QA' it just does an index scan and there is no reference to the table data pages whatsoever. This is because QA' is referring only to the primary keys of tables – `n_nationkey` for NATION and `n_regionkey` for REGION – which have indexes on them. The optimizer detects this and therefore answers the query from the index itself. SIMCHECK also detects this difference because the *Index Flag* feature is set for QA' but not for QA.

A completely opposite scenario to the one just described is seen in the pair of queries, QA and QA". Here the queries are very different in terms of their structure but the templates of the plans generated by the DB2 optimizer are exactly the same! The reason for this is that the effective selectivities and input sizes of the tables involved in both the queries are similar. Also, the queries are structurally similar. SIMCHECK is successful in identifying these aspects and therefore predicts the same plan for QA" as that used for QA.

6 Query Clustering

This section describes the algorithm that is used by PLASTIC for clustering the queries. The major issues related to clustering queries are: (1) dealing with a large collection of queries; (2) deciding the number of clusters to be formed; and (3) ensuring the accuracy of clusters.

PLASTIC uses a clustering technique that is based on the *Leader* algorithm proposed by Hartinger [9]. In the Leader algorithm, whenever a pattern does not find a match with any of the existing clusters, a fresh cluster is created with the pattern becoming its representative or "leader". Our choice of Leader is based on several factors. First, it is an O(kn) clustering algorithm, where k is the number of clusters and n is the number of queries, which is attractive for online environments that deal with large query workloads. This time complexity is because every query needs to be compared with k cluster leaders in order to assign it to a cluster. In practice, $k << n$ since every leader potentially represents hundreds of queries, making Leader a linear algorithm with respect to the total number of user queries. Further, Leader is also incremental in the sense that when a new query is included in the Cluster Database, it does not require a reworking of the existing clusters.

We could also have chosen alternative algorithms such as Nearest Neighbor (NN) to compute the clusters, but since these algorithms typically require all queries to be compared with all other queries, they need to access the same query multiple times and for a large set of queries, this may become very expensive. That is, the complexity of such an NN-clustering algorithm is O(n^2) which when compared with the O(kn) of Leader clearly indicates that Leader would prove superior for large query workloads.

The second reason to choose Leader is that it is extremely simple to manage. In comparison, an algorithm like BIRCH [18], which is also linear and incremental, has to incur the overhead of managing a complex data structure called the CF-TREE.

On the down side, however, the clusters formed by Leader are a function of the *query sequence* since all unmatched queries become cluster representatives. That is, cluster formation is based on the order in which queries arrive. However, the order-dependence does not have an adverse effect on cluster accuracy since that depends only on the cluster adius which is determined by the Threshold setting. It is only the *efficiency* of cluster identification that may become lower because a larger number of clusters than strictly necessary are generated due to a particular query order.

We now discuss a few characteristics of the clusters formed by the Leader algorithm. Firstly, the distance function $dist_{ij}$ produces clusters that are hyper-ellipsoids in the query template space. This is because the stretch in a particular dimension depends on the table in that particular dimension, and since every $dist_ij$ has a different value in the denominator the stretch of the cluster in that particular dimension will be different, making the cluster an ellipsoid. Secondly, in our current implementation, the leaders are searched for the first acceptable fit, not the best one. This is done for the sake of simplicity, but if required, it is easy to modify the algorithm to implement a best fit policy.

With respect to the issue of deciding the *number* of clusters, this is dictated by the Threshold setting. The smaller the threshold, the more the number of clusters and the lesser the error probability. Another solution is to specify a maximum number of clusters and then, when this limit is reached, assign each new query to whatever happens to be the nearest existing cluster. Modifying the Leader algorithm to implement this policy is straightforward.

If the same plan applies to the complete region covered by a cluster then there will be no error in plan prediction if a new query is mapped to this cluster. It is only if the cluster's space is covered by more than one plan, that there will be an error in prediction because all the queries mapping to this cluster will be assigned the plan associated with the query leader. The error involved in such an assignment will increase as the difference in effective table sizes between the new query and the leader increases. If the cluster regions are large then the stretch of clusters that straddle plan boundaries is large and hence the error in prediction may be higher. This aspect is controlled by the appropriate setting of the Threshold value.

It should be noted here that our algorithm may map more than one cluster to a single plan. This is essentially because the clusters generated by the algorithm are of a fixed size which is tuned by the user. Thus, filling up the complete region in the plan space where a plan P_i is applicable may require a large number of clusters. For a set of k clusters, this imposes a space overhead of $k-1$ query feature vectors and $k-1$ plan templates. But this can be largely overcome if we compare the plans corresponding to the leaders and point all matching cluster leaders to the same physical plan P_i.

7 Classification of New Queries

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. For example, we can use 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 leader is encountered.

An obvious question that arises with this approach is that the number of clusters may become very large and matching with every leader may therefore become an expensive affair. Accordingly, we need a faster technique such as decision trees or hierarchical clustering [4] – 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, Predicate Counts 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.

In the current setup of PLASTIC, the classical C5.0 decision tree induction algorithm [12] is used to generate the decision trees. The C5.0 algorithm chooses features for splitting in an order that provides the maximum information gain at every split.

Table	Cardinality	Size (in MB)
PART	200000	29.8
PARTSUPP	800000	124.7
CUSTOMER	150000	26.6
SUPPLIER	10000	1.7
LINEITEM	4859686	658.6
NATION	25	$2 * 10^{-3}$
REGION	5	$4 * 10^{-4}$

Table 3: TPC-H Table Statistics

8 Performance Framework

PLASTIC has been evaluated on the TPC-H Database populated at scale 1 (i.e. 1 GB of data). The tables present in the database and their sizes are given in Table 3. The database was populated using DBGEN [1] and the queries were generated using QGEN [1] – these queries were modified to result in un-correlated SPJ queries. While we conducted experiments with a variety of TPC-H queries, we discuss the results here for only a few representative queries due to space limitations.

8.1 Metrics

We assume in our experiments that the queries received by the system will be independent and uniformly distributed over the feature space. The metrics used for evaluation are as follows:

8.1.1 Accuracy

We measure accuracy in terms of what is referred to as an *empirical* or *training* risk in the machine learning literature. In our case it is the ability of the cluster boundaries to discriminate the plan boundaries sketched by an optimizer. In this regard, we consider a cluster *ambiguous* if there is more than one optimizer plan applicable to different queries in the cluster. The predicted plan for a new query is the plan mapped to the leader. But there may be other queries in the cluster which may not have this plan as their optimizer-generated plan. The expected probability of a query being assigned to the wrong plan is given by:

$$E(q) = \Sigma_i P_{error}(q/C_i)) * P(C_i)$$

where C_i is an ambiguous cluster, $P(C_i) = \frac{1}{TotalNumberofClusters}$ is the apriori probability that the query is assigned a particular cluster (assuming each cluster is equally likely), and $P_{error}(q/C_i)$ = Probability that q is classified incorrectly if assigned to cluster C_i. In our experience, ambiguous clusters typically overlap two plans, and therefore $P_{error}(q/C_i)$ can be approximated to 50%.

8.1.2 Efficiency

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.

8.1.3 Risk Factor

We define Risk Factor as the maximum risk involved in predicting plans using PLASTIC. This risk is computed as the worst case extra cost incurred when PLASTIC does not choose the optimizer-generated plan for a particular query. The worst case occurs when a leader is located at a point in the query template space where the optimizer switches from one plan to another, and the incoming query is located at the periphery of this leader's cluster.

8.1.4 Space Overhead

The space overhead involved is defined in terms of the amount of space required for storing a single query. The overall storage can be estimated using this value. The space overhead is proportional to the accuracy desired. The higher the accuracy desired, the greater is the space overhead involved and vice versa.

8.2 SIMCHECK Algorithm Configuration

The SIMCHECK algorithm requires three constants: w_1, w_2 and $Threshold$. Through considerable empirical evaluation, we have found that the values 0.7 and 0.3 for w_1 and w_2 give satisfactorily accurate results. Further, these settings are robust to the extent that performance is only marginally impacted by varying these weights to $\pm$ 0.1.

With regard to the Threshold, we found that its ideal value is a function of the nature of the underlying database. For databases with large tables, lower threshold values were found to be appropriate since the optimizer is more sensitive with respect to selectivity changes in such environments. Specifically, for the TPC-H database, we found that setting the Threshold to 0.01 was a good choice. This is especially true for higher selectivities where if we generate large clusters due to a high threshold value, the risk involved will be considerable. Ideally, there should be smaller clusters towards the high selectivity regions and larger towards the low selectivity regions. Since in our current implementation, we are using uniform sized clusters for simplicity, the threshold needs to be set low so as to minimize the risk for the high selectivity regions.

Metric	DB2	P-DB2	
		Leader	Decision Tree
Accuracy	100%	90.76%	88.8%
Efficiency	0.1s (avg. case)	0.004s (worst case)	0.00025s
Space	–	1.97 KB	3.96KB

Table 4: Performance Profile for Query Q2'

8.3 Evaluation Testbed

All our experiments were conducted using the optimizer of DB2 Universal Database Version 7 on a Windows 2000 Workstation running on a Pentium III-800 MHz machine with 256 MB RAM, and 20 GB disk. DB2 gives a choice of 9 optimization classes – we use the default optimization class, 5, in our experiments.

9 Experimental Results

Our first experiment evaluates the performance for a query workload that is generated from the same query template, while the second experiment investigates the sharing of plans among queries that arise from different query templates. These experiments are described in the remainder of this section.

9.1 Experiment 1: Intra-query Plan Sharing

We consider here query Q2' for which we have already seen the plan diagram and associated cluster diagram in Figures 1 and 2, respectively. There were 65 clusters created for this experiment and the associated performance numbers are shown in Table 4, where DB2 refers to the traditional optimizer performance while P-DB2 refers to the performance of a DB2 that has been augmented with PLASTIC. We evaluate the performance of P-DB2 under two environments: First, where the Leader algorithm is used for both classification and clustering, and second, a more sophisticated implementation where a decision tree is built on the clusters and used for classification.

The first point to note in Table 4 is that P-DB2 delivers an accuracy of around 90 percent, which shows that PLASTIC can be profitably used in conjunction with a traditional optimizer. Further, note that in this formulation, we have assumed that the DB2 optimizer always chooses the best plan and hence its accuracy is quoted as 100%. But in general, DB2 does not give as good a plan at level 5 optimization as the one it gives at level 9 optimization. A deterrent to always run the optimizer at level 9 is the fact that this level involves considerable additional computation cost. However, with P-DB2, 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

Ambiguous Cluster	DB2 Cost	P-DB2 Cost	Risk Factor (in %)
1	261209	266260	1.9
2	241054	246000	2
3	173913	188684	1.1
4	158577	158681	0
5	161814	159078	-.02

Table 5: Risk Factor Analysis for Q2'

best optimization level, which would have been difficult to achieve otherwise.

Where P-DB2 really scores over DB2 is with regard to Efficiency, that is, the optimization time. On average, DB2 takes 0.1s for optimization while P-DB2, even when using the Leader algorithm, which does a brute force search takes only 0.004s in the *worst case* (which occurs when it has to compare with all the 65 cluster leaders) ! When the search is accelerated through a decision tree classifier, the optimization times further reduce by an order of magnitude. However, this improvement comes at the cost of errors induced by the decision tree algorithm, resulting in a marginally decreased accuracy.

Moving on to the risk that is incurred when P-DB2 *does* make wrong choices, the risk values for 5 ambiguous clusters are shown in Table 5 (the measurement is in *timerons* which is the DB2 unit for cost estimation and represents a weighted sum of IO and CPU cost). From these statistics we see that there is only a a nominal risk associated with each incorrect assignment. This can be traced to the fact that when PLASTIC errs in cluster selection, it is found to choose the second best plan in the majority of the cases. More importantly, PLASTIC errs only when a cluster overlaps different plans. This means that the switching between these plans falls in that cluster and at switching points the costs of the associated plans are *very close to each other*. Therefore, the impact of the error is close to negligible. In fact, for ambiguous cluster 5, PLASTIC has happened to serendipitously pick up an even better plan than DB2 (due to its running only at level 5, whereas a higher level of optimization may have recommended the plan output by PLASTIC).

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*.

9.2 Experiment 2: Inter-query Plan Sharing

This experiment investigates a totally different dimension to the use of PLASTIC and is aimed to highlight the fact that PLASTIC identifies similarities at a higher plane than mere structural mapping. We have already seen two examples demonstrating this in Section 5. This section analyzes this capability of PLASTIC in more detail and presents four different cases where queries having significantly different structures and statistics are correctly optimized by P-DB2.

9.2.1 Different Projection Attributes

Given below is a query where certain projection attributes that were present in Q2' are removed and new attributes have been added[2], keeping the rest of the query the same.

```
select
    s_name, n_name, p_partkey, p_mfgr,
    s_address, s_phone, s_comment
from
    part p, supplier s, partsupp ps,
    nation n, region r
where
    ... (rest same as Q2')
```

The plan generated by DB2 for this query is the same as that for Q2'. This is because DB2 does not change its plans unless there is a considerable impact of the difference in projection attribute sizes on the effective join sizes of individual tables. For example, if the projection attributes are changed to * in this query, resulting in a very wide tuple, DB2 changes its plan choice.

PLASTIC is also able to correctly identify the plan similarity for the above query. It becomes possible because we do not match attribute or table names but rely only on the structural and statistical features. Since the queries are structurally similar and their statistical distance is within the similarity threshold, their plans are estimated to be the same.

9.2.2 Different Selection Predicates

From the above discussion, it is straightforward to conclude that even if we change/add/remove selection predicates in a query, the optimizer may not change the plans. All that matters for the optimizer are the estimated sizes calculated on the basis of predicate selectivities and whether these predicates are SARGable. For example, consider the query shown below which augments Q5', our SPJ version of the Q5 query from the TPC-H benchmark, with the additional selection predicate (R_COMMENT > 'HELLO').

```
select
     l_extendedprice, l_discount
from
     customer, orders, lineitem,
 supplier, nation, region
where
     c_custkey = o_custkey
     and l_orderkey = o_orderkey
     and l_suppkey = s_suppkey
     and c_nationkey = s_nationkey
     and s_nationkey = n_nationkey
     and n_regionkey = r_regionkey
     and r_name='AFRICA'
     and R_COMMENT > 'HELLO'
     and o_orderdate >=
            date ('1997-01-01')
     and year(o_orderdate ) <
            (year('1997-01-01') + 1);
```

[2] The bind variables have the same values in both queries.

In this case, PLASTIC correctly recognizes that DB2 will eventually choose the same plan for both Q5' and the augmented version since the overall selectivities of the REGION table do not change to the extent that it will affect the plan selection of the optimizer.

9.2.3 Different Join Attributes

Given below is another variant of query Q5' where the second predicate of Q5' has been changed to "l_commitdate=o_orderdate" from the original "l_orderkey=o_orderkey".

```
select
    l_extendedprice, l_discount
from
    customer, orders, lineitem,
    supplier, nation, region
where
    ...
    L_COMMITDATE=O_ORDERDATE
    ...
```

DB2 generates the same plan for both this query and a variant where the aforementioned commit_date join predicate is replaced with "l_shipdate = o_orderdate". PLASTIC also identifies this because the join index and predicate counts of both queries are the same. This is an example of a situation where checking for equality on JIC values plays an important role.

9.2.4 Different Tables in the Query

Our final experiment investigates the biggest difference between two queries, namely, changing the query *tables* themselves. For this we created a new table NATION_1 which was different in terms of number of tuples, number of attributes, and column sizes of the attributes, from the original table NATION of the TPC-H benchmark. The cardinalities of the tables were 25 and 36 tuples, respectively. DB2 produces the same plan for Q5', run once with NATION and then with NATION_1. The point we are making here is that if the overall sizes of tables and certain characteristics of the tables, such as primary keys and their indexes, etc. are similar, the optimizer's choices are not subject to much change. Obviously, table and attribute names do not matter for an optimizer. PLASTIC also incorporates a similar logic and therefore successfully predicts that the two queries will have the same plan.

10 Related Work

Techniques such as *multi-query optimization* (MQO) [13, 14, 16, 11, 8] and *parametric query optimization* (PQO) [5, 3, 10], have been previously proposed for enhancing the query optimization process. Both these techniques are inherently computationally hard – for example, the search space in MQO is doubly exponential in the size of the queries. This has led to the design of heuristic-based solutions, such as those presented in [13].

Our approach is fundamentally different from MQO in that we do not attempt to *optimize* queries but merely to make effective use of the *results* of prior optimizations. Moreover, while we do group queries into clusters, the plan selection is applicable on a per-query basis and is therefore not restricted to query batches. Finally, our optimization is not limited to a temporal window of queries, but can be utilized across widely dispersed query sets.

Moving on to PQO, its coverage of the query space is typically an offline process. In contrast, our approach can be implemented in either an offline manner where artificial queries are generated so as to create clusters that cover the query space, or more practically as an online process with regard to both cluster formation and query plan selection. That is, the query space can be covered incrementally on demand when user queries arrive at the database system.

Another significant difference with PQO is that our plan selection process involves only the traversal of a simple decision tree, whereas PQO requires a *spatial* storage and indexing mechanism. This is because the scheme requires storing not only the set of optimal plans but also the *regions* in which each of these plans are optimal. Even for simple linear cost functions, the shapes of these regions turn out to be convex polyhedrons [5], mandating spatial storage and access in order to identify which plan is to be utilized for a newly arrived query. This issue assumes importance since supporting spatial databases is well-known to be an expensive proposition [17].

Finally, while PQO is concerned with completely characterizing the plan space for a given query, our approach extends to *sharing* of plans across similar queries.

10.0.5 Comparison with Modern Optimizers

Some modern optimizers also provide plan reuse facilities. We discuss Oracle9i Optimizer [2] here and how adding PLASTIC, to any such optimizer, can augment its capabilities. The Oracle database system provides a mechanism, called "stored outlines", for preserving queries and execution plans. When the system parameter USE_STORED_OUTLINES is set to true, the optimizer compares the incoming query with the stored queries and if an *identical* match is found, the associated plan is used.

The point to note here is that the query matching is done at the *syntactic* level. There is a one-to-one correspondence between SQL text and its stored outline. If a different literal is specified in a predicate, then a different outline applies. To avoid this, Oracle also allows bind variables to be used instead of constants to allow a wider coverage. This approach is still somewhat limited in several ways. Firstly, the query matching is very strict – a slight change in the structure of query, for example, adding or replacing of a projection attribute, will result in the optimizer not utilizing the existing plan. Secondly, it does not take into account the fact that several selection predicates on a particular table can together generate a selectivity for the table which is similar to that of a previously stored query. This is essentially what was illustrated by the example involving QA and QA" in Section 5. Thirdly, a more serious problem is that the query plan is the same for the *complete range of values* of a bind variable since Oracle adopts the heuristic of assuming small values for the selectivity of bind variable-based predicates. Specifically, it chooses a selectivity of 0.05 for all range predicates associated with bind variables, a heuristic that can prove very costly for database environments with higher selectivity values. Our approach, on the other hand, tries to address all these three issues in a much more flexible and fine-grained manner.

It should thus be noted that PLASTIC does not just map a parametric space based on changes in bind variables of selection predicates but works at the level of sharing *between* queries, a feature we expect would be desirable in practice.

11 Conclusions

We have presented and evaluated PLASTIC, a value-addition tool for query optimizers that attempts to efficiently and accurately predict, given previous training instances, what plans would be chosen by the optimizer for new queries. Apart from the obvious advantage of speeding up optimization time, it also improves query execution efficiency since it makes it possible for optimizers to always run at their highest optimization level 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.

PLASTIC's design is based on clustering in the query space, rather than in the plan space, and its query feature vector includes a variety of structural and statistical attributes. This framework allows PLASTIC, unlike the state-of-the-art in commercial optimizers, to identify query similarity at a much broader level, including handling changes in projection, selection and join predicates, as also in the query tables themselves. Therefore, it promises to significantly improve the utility of plan cacheing.

A performance evaluation of PLASTIC against various TPC-H benchmark queries showed high plan prediction accuracy, an improvement by an order of magnitude in optimization time, a nominal error penalty and a low space overhead.

In our future work, we plan to extend PLASTIC, which currently only handles basic SPJ queries, to also support nested queries, groups and aggregates. We also plan to investigate the design of *variable-sized* clusters which will facilitate tuning cluster sizes to match the plan volatility in each region of the feature space. Finally, we are working on automated mechanisms for setting the weights and threshold values used in the similarity checking algorithm.

Acknowledgements

We thank S. Sudarshan and S. Sarawagi of IIT Bombay for their encouragement and advice during this work.

References

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

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

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

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

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

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

[7] A. Ghosh, J. Parikh, V. Sengar and J. Haritsa, "Query Clustering for Plan Selection", Tech Report, DSL/SERC, Indian Institute of Science, July 2002.

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

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

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

[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] R. Quinlan, *http://www.rulequest.com/see5-info.html*

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.