Distributed and Parallel Databases, 6, 373-- 420 (1998)

Solving Local Cost Estimation Problem for Global Query Optimization in Multidatabase Systems

QIANG ZHU qzhu@umich.edu
Department of Computer and Information Science, The University of Michigan - Dearborn, Dearborn, MI 48128, USA

PER-AKE LARSON * palarson@microsoft.com
Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Abstract. To meet users' growing needs for accessing pre-existing heterogeneous databases, a multidatabase system (MDBS) integrating multiple databases has attracted many researchers recently. A key feature of an MDBS is local autonomy. For a query retrieving data from multiple databases, global query optimization should be performed to achieve good system performance. There are a number of new challenges for global query optimization in an MDBS. Among them, a major one is that some local optimization information, such as local cost parameters, may not be available at the global level because of local autonomy. It creates difficulties for finding a good decomposition of a global query during query optimization. To tackle this challenge, a new query sampling method is proposed in this paper. The idea is to group component queries into homogeneous classes, draw a sample of queries from each class, and use observed costs of sample queries to derive a cost formula for each class by multiple regression. The derived formulas can be used to estimate the cost of a query during query optimization. The relevant issues, such as query classification rules, sampling procedures, and cost model development and validation, are explored in this paper. To verify the feasibility of the method, experiments were conducted on three commercial database management systems supported in an MDBS. Experimental results demonstrate that the proposed method is quite promising in estimating local cost parameters in an MDBS.

Keywords: multidatabase, global query optimization, cost model, query sampling, multiple regression

1. Introduction

A *multidatabase system (MDBS)* integrates data from multiple pre-existing databases managed by heterogeneous *component (local) database systems (DBS)* in a distributed environment. The MDBS can only interact with component DBSs at their **external** user interfaces. A key feature of an MDBS is the *local autonomy* that individual DBSs retain to serve existing applications [2]. Briefly speaking, local autonomy refers to the situation where each component DBS retains complete control over its local data and operations.

The term MDBS has been used by different people to mean different things [20]. Litwin *et al.* [11] use it to mean a system managing multiple databases without a global schema. The essential component of such an MDBS is a language used to manage interoperable databases. Local database systems are loosely coupled. Local autonomy is fully supported. Dayal and Hwang [6], Breitbart and Silberschatz [1], Lu et al. [13], and many others use

* Current address: Microsoft Corporation, One Microsoft Way, Redmond, WA 98052--6399, USA

the term MDBS to mean a system managing multiple databases with one or more global schemas. The essential component of such an MDBS is a global system built on top of the component DBSs to provide global query optimization, transaction management, and other global services. Local database systems are more tightly coupled than the previous model. Local autonomy is supported to certain degree. In this paper, we adopt the latter MDBS model.

A user who interacts with an MDBS is called a *global user*. A user who interacts with a component DBS in the MDBS is called a *local user*. An MDBS provides a global user with a simple and consistent view of database access. A global user can issue a global query for retrieving data from several component databases. To achieve good overall system performance, *global query optimization* is needed.

Local autonomy poses new challenges for global query optimization in an MDBS [13, 20, 24]. Among them, the crucial one is that some local information needed for global query optimization, such as local cost parameters, may not be available at the global level. Because of local autonomy, a component DBS may not expose all its information at the global level. Lack of local information increases the difficulty for the global query optimizer to choose a good strategy for executing a global query. The global query optimizer needs some local information to decide how to decompose a global query into component queries and where to execute the component queries. To perform global query optimization, the problem of how to estimate local cost parameters in an MDBS needs to be solved.

Recently, several researchers have been investigating how to solve this crucial problem for global query optimization in an MDBS. In general, if an autonomous component DBS is viewed as a black box whose optimization information is hidden from the global optimizer, there are three potential approaches to obtain or estimate local optimization information [26, 28]:

- performing some testing queries to test the black box;
- guessing necessary information subjectively based on external characteristics of and previous knowledge about the black box;
- monitoring the behavior of the black box at run time and dynamically collecting necessary information.

The third approach is actually a type of adaptive query optimization that can be implemented by borrowing existing adaptive query optimization techniques in the literature. Lu and Zhu [12, 24] discussed some issues for applying this approach to an MDBS.

The first two approaches are relatively new and were investigated recently. In [7], Du *et al.* proposed a calibration method to deduce necessary local information. This method belongs to the first approach. The idea is to construct a local synthetic calibrating database (with some special properties), and then run a set of special queries against this database. The access method used for executing such a query is known because of the special properties of the database and query. Cost metrics for the queries are recorded and used to deduce the coefficients in the cost formulas for the access methods supported by the underlying local database system by using the properties of the database and queries.

In [27], we introduced an alternative method, called the query sampling method, which also belongs to the first approach. The idea is to classify queries according to their potential access methods to be used, then perform sample queries against the actual underlying database, and use observed costs to derive a local cost formula for each query class by multiple regression. Since the actual underlying database rather than a synthetic database is used in the method, the derived cost formulas are expected to reflect the performance of the real user environment better.

In [24], Zhu suggested to perform carefully-designed probing queries on a component database to probe and deduce some optimization information. In fact, sample queries can be considered as a special type of probing queries. In [25], Zhu discussed the issues faced in estimating selectivities of queries in an MDBS environment.

In [26, 28], we also proposed a fuzzy optimization method which belongs to the second approach. The idea is to build a fuzzy cost model based on experts' knowledge, experience and guesses about the required optimization parameters and perform fuzzy query optimization based on the fuzzy cost model.

Our results about the query sampling method reported in [27] were preliminary. For instance, the query classification rules, cost formulas and statistical procedures presented there were simplistic. In this paper, we will fully explore and further extend this method. More query classification rules, new sample queries, and a new statistical procedure to automatically generate a suitable cost formula for a query class are proposed. The feasibility of the method was verified experimentally on three commercial database management systems (DBMS). This method was designed for an MDBS prototype, called CORDS-MDBS, that was developed jointly by the University of Waterloo and the Queen's University.

Although a number of sampling techniques have been applied to query optimization in the literature, all of them perform *data sampling* (i.e., sampling data from databases) instead of *query sampling* (i.e., sampling queries from a query class). Muralikrishna and Piatetsky-Shapiro *et al.* [14, 19] discussed how to use data sampling to build approximate selectivity histograms. Hou and Lipton *et al.* [8, 10] investigated several data sampling techniques, e.g., simple sampling, adaptive sampling and double sampling, to estimate the size of a query result. Olken *et al.* [16] considered the problem of constructing a random subset of a query result without computing the full result. All their work is about performing a given query against a sample of data and deriving properties for the underlying data (e.g., selectivities). The query sampling method presented in this paper considers performing a sample of queries against the (entire) underlying database and deriving a property about a query performed on the underlying DBMS, i.e., performance of the query on the DBMS.

The rest of this paper is organized as follows. Section 2 will introduce the key idea and assumptions of the query sampling method. To carry out the query sampling method, one has to solve the following three main problems: (1) how to classify queries, (2) how to draw sample queries, (3) how to derive satisfactory cost formulas. These issues will be discussed in Sections 3 $\sim$ 5, respectively. Section 6 will report our experimental results to show the feasibility of the query sampling method. Section 7 will summarize the conclusions and list some future research issues.

2. Methodology

To solve the incomplete local information problem in an MDBS, this paper presents a query sampling method that is based on statistical sampling and regression analysis techniques. Before describing the idea of the method, let us first make some assumptions.

2.1. Assumptions and Notations

2.1.1. Considered query set Different component DBMSs may adopt different local data models. At the global level of an MDBS, there usually is a common global data model. In our MDBS prototype, the global data model is assumed to be relational. Each component DBMS is associated with an MDBS agent which provides a relational interface if the component DBMS is non-relational. Hence, the global query optimizer in the MDBS may view all participating component DBMSs as relational ones.

Many possible queries can be issued against a component database. Since most common queries can be expressed by a sequence of select (σ), project (π) and join ($\bowtie$), only these three types of operations (so-called ''work-horse'' operations in the relational model) are considered in this paper. The cost of a query composed from these operations can be estimated by composing the costs of the operations. Aggregates are not considered in this paper. However, the ideas and methods to be discussed in this paper can be directly extended to handle a query with aggregates. In real systems a project is usually computed together with the select or join that it follows, so we will not consider it separately. A select that may or may not be followed by a project is called a *unary query*. A join that may or may not be followed by a project is called a *join query*. Since the majority of practical joins are equijoins, only equijoins are considered. The method can be easily extended to handle general joins.

Let G be the set of all component (unary and join) queries that can be issued against a component database DB. Let R_i $(1 \leq i \leq K)$ denote a table in DB; $\alpha^{(i)}$ denote a non-empty list of columns in R_i $(1 \leq i \leq K)$; $\alpha^{(ij)}$ denote a non-empty list of columns in R_i and R_j $(1 \leq i, j \leq K)$; $F^{(i)}$ denote the qualification of a query on R_i; $F^{(ij)}$ denote the qualification of a query on R_i and R_j; $R_i.a_n$ $(n \geq 1)$ denotes a column of R_i (the prefix R_i can be omitted if no ambiguity); $C^{(i.a_n)}$ be a constant in the domain of column $R_i.a_n$ (the superscript is sometimes omitted if no confusion). Without loss of generality, the qualifications of queries are assumed to be in conjunctive normal form. The *basic predicates* allowed for unary queries are of the form $R_i.a_n\ \theta\ C^{(i.a_n)}$, where $\theta \in \{=, \neq, >, <, \geq, \leq, nil\}$. $R_i.a_n\ nil\ C^{(i.a_n)}$ stands for the '$true$' *(empty) predicate*, also called a *dummy basic predicate*. The basic predicates allowed for join queries are of the forms $R_i.a_n = R_j.a_m$ as well as $R_i.a_n\ \theta\ C^{(i.a_n)}$. Since we only consider equijoin queries, each join query has at least one conjunct $R_i.a_n = R_j.a_m$. Let $\wedge$ and $\vee$ denote the logical connectives AND and OR, respectively, and $|X|$ denote the cardinality of set X.

2.1.2. Characteristics of cost estimation for query optimization In order to perform global query optimization in an MDBS, costs of component queries need to be estimated. The global query optimizer decides where to perform component queries and how to transfer intermediate results among local sites, based on the estimated costs. For the estimated costs, the more accurate, the better. However, there are several relaxed characteristics of cost estimation for query optimization, three of which are discussed as follows.

The first relaxed characteristic is that the required precision of estimation is not as high as in many engineering applications. In other words, estimation errors can be tolerated to certain degree. The reason for this is that the goal of most practical query optimizers is to find a good execution plan for a query rather than a truly optimal execution plan for the query. As long as an estimated cost and a real cost are of the same order of magnitude, the estimated cost is usually acceptable. For example, if the costs of two execution plans for a query are of the same order of magnitude, such as several seconds, or several thousands of seconds, the two plans can be considered of the same goodness because one is not much better than the other. However, if the cost of one plan is several seconds, while the cost of another is several thousands of seconds, the former is obviously much better than the latter.

Another relaxed characteristic of cost estimation for query optimization is that it is usually satisfactory if most estimated costs are within an acceptable error range. In other words, it is tolerated if a few estimated costs have unacceptable errors. The reason for this is that a practical query optimizer is said to be good if it can improve the performance of most queries, not necessarily all queries.

The last relaxed characteristic is that the accuracy of estimated costs for frequently-used queries is more important than the accuracy of estimated costs for rarely-used queries. This implies that we can sacrifice the latter to guarantee the former if it is hard to satisfy both due to estimation overhead, complexity and/or other reasons.

2.2. *Idea of Query Sampling Method*

The main idea of the query sampling method to be discussed in this paper is described in Figure 1. The first step of the method is to group all possible queries on a component database into more homogeneous classes so that the costs of queries in each class can be estimated by the same formula. This can be done by classifying queries according to their potential access methods. A sample of queries are then drawn from each query class and run against the actual component database. The costs of sample queries are recorded and used to derive a cost formula for the queries in the query class by multiple regression. The coefficients of the cost formulas for the component database system are kept in the multidatabase catalog and retrieved during query optimization. To estimate the cost of a query, the query class to which the query belongs needs to be identified first, and the corresponding cost formula is then used to give an estimate for the cost of the query.

To classify queries, a number of relevant issues need to be studied, such as what useful information is available for classification, what rules can be used for classification, and what procedure can be applied to classification. To draw sample queries from query classes, several relevant issues should be investigated, such as what query sampling approach is appropriate, what sampling procedures can be used for different query classes, and how to get good sampling procedures. To derive cost formulas for query classes, some relevant issues must be considered, such as what variables can be included in a cost formula, what

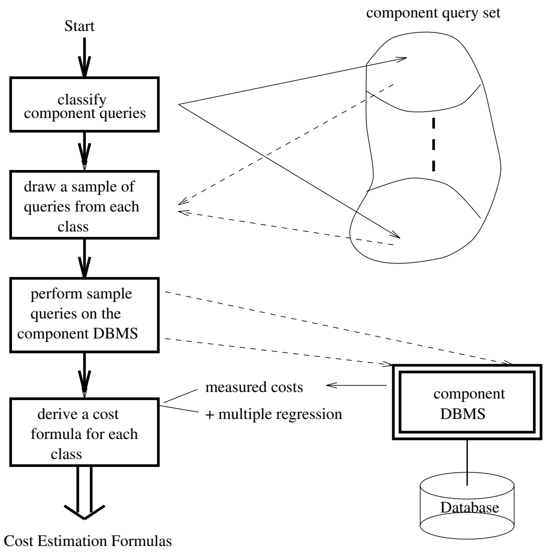

Figure 1. The Idea of Query Sampling Method

cost formulas are suitable for different query classes, how the goodness of a cost formula can be evaluated. All these issues are to be discussed in the following sections.

3. Classification of Queries

The first step of the query sampling method is to classify the set of component queries that can be performed on a component DBS. The objective is to group queries into more homogeneous classes so that the costs of queries in each query class can be estimated by the same formula. This section discusses how to obtain such a query classification.

3.1. Useful Information for Classification

The costs of queries executed by using the same access method, such as a sequential scan or a nested-loop join, can be estimated quite accurately by the same formula. Therefore, it is a good principle to classify queries according to their employed access methods. However, which access method to be used for a component query may not be known at the global level in an MDBS. It depends on the underlying component DBMS.

Fortunately, there are some common rules for choosing an access method in many DBMSs. Based on these common rules and some other available information, we can group queries into more homogeneous classes. The costs of all queries in one class are estimated by the same formula. If available information is sufficient, we can classify queries in such a way that each class corresponds to one access method. The estimated costs are expected to be quite accurate in this case. If the available information is not sufficient, it is possible that queries executed by different access methods are put in the same class. Since the practical goal of query optimization in an MDBS is the same as those of many traditional query optimizers, that is, avoiding bad execution plans instead of achieving a truly optimal one, estimation errors can be tolerated to some degree.

Unlike a query optimizer in a traditional DDBS, the global query optimizer in an MDBS has limited available information. To classify queries, the following types of information can be exploited:

- *characteristics of queries*: such as query types (unary or join), the number of conjuncts, types of predicates and so on. This type of information can be obtained by analyzing the syntax of a given query.
- *characteristics of operand tables*: such as cardinality of a table, the number of columns, indexed columns, clustered-indexed columns and so on. This type of information can usually be obtained from the multidatabase catalog or local catalogs.
- *characteristics of underlying component DBMSs*: such as types of supported access methods and maybe rules for choosing an access method for a query. This information can be obtained from the documentation of a component DBMS, such as DBA's (database administer) guidance.

The above types of information are generally available from most DBMSs, hence we assume that they are the minimum available information at the global level in an MDBS. Some other types of information, such as (estimated) sizes of result tables, can also be used for classification. Some component DBMSs may provide additional information, such as the execution plan generated for a query. The additional information can be used to refine a classification, resulting in more accurate cost estimates.

3.2. Common rules

There are some common rules adopted in many DBMSs for choosing an access method for a query. These common rules can be used to classify queries. For example, the following common rule is obviously true for all DBMSs:

r_1: *a unary query and a join query are executed by using different access methods.*

Based on this rule, we can put unary queries and join queries into two separate classes. Some other common rules are listed in Tables 1 and 2.

A common rule, such as r_2, is said to be inapplicable if either the underlying DBMS does not support the relevant access method or the qualification of the query does not have a required conjunct. A predicate $R_i.a_n \ \theta \ C$ ($\theta \in \{=, <, >, \leq, \geq\}$) is said to be *index-usable* if $R_i.a_n$ is (clustered or non-clustered) indexed. A conjunct is said to be index-usable if every predicate in the conjunct is index-usable. For example, the conjunct $[R_1.a_1 < 3 \ \vee \ R_1.a_2 = 9 \ \vee \ R_1.a_3 \leq 2]$ is index-usable if all the predicates involving $R_1.a_1$, $R_1.a_2$ and $R_1.a_3$, respectively, are index-usable.

Each common rule in Tables 1 and 2 specifies what queries are most likely to be executed by the same access method. Therefore, the qualified queries for each common rule can be put into the same query class.

Table 1. Some Common Rules for Unary Queries

Label	Description of Common Rules
r_2	a clustered-index scan method with a key value is usually chosen if the qualification of a query has at least one conjunct $R_i.a_n = C$ where $R_i.a_n$ is clustered-indexed
r_3	an index scan method with a key value is usually chosen if r_2 is inapplicable and the qualification of a query has at least one conjunct $R_i.a_n = C$ where $R_i.a_n$ is indexed
r_4	a clustered-index scan method with a range is usually chosen if r_2 and r_3 are inapplicable and the qualification of a query has at least one conjunct $R_i.a_n\ \theta\ C$ where $R_i.a_n$ is clustered-indexed and $\theta \in \{<, \leq, >, \geq, \}$
r_5	an index scan method with a range is usually chosen if $r_2 \sim r_4$ are inapplicable and the qualification of a query has at least one conjunct $R_i.a_n\ \theta\ C$ where $R_i.a_n$ is indexed
r_6	a sequential scan method is usually chosen if $r_2 \sim r_5$ are inapplicable

Table 2. Some Common Rules for Join Queries

Label	Description of Common Rules
r_7	a clustered-index join method is usually chosen if the qualification of a query has at least one conjunct $R_i.a_n = R_j.a_m$ where either $R_i.a_n$ or $R_j.a_m$ is clustered-indexed
r_8	an index join method is usually chosen if r_7 is inapplicable and the qualification of a query has at least one conjunct $R_i.a_n = R_j.a_m$ where either $R_i.a_n$ or $R_j.a_m$ is indexed
r_9	a nested-loop join method with reduction on operand table(s) via index(es) is usually chosen if r_7 and r_8 are inapplicable and the qualification of a query has at least one index-usable conjunct for R_i or R_j
r_{10}	a sort-and-merge join method is usually chosen if $r_7 \sim r_9$ are inapplicable

These common rules are valid for many DBMSs. However some of them may need to be adjusted for a specific DBMS because an actual rule for choosing an access method in the DBMS may not be consistent with one or more common rules discussed above.

For example, a DBMS may choose a sequential scan method if $r_2 \sim r_4$ are inapplicable and the qualification of a given query has at least one conjunct $R_i.a_n\ \theta\ C$ where $R_i.a_n$ is indexed and $\theta \in \{<, >, \leq, \geq\}$. This rule is inconsistent with common rule r_5 in Table 1. This rule can then be merged with rule r_6 to get a new rule r_5'. We may, therefore, replace r_5 and r_6 by this actual rule r_5' when we classify queries for the DBMS.

In fact, if we keep using r_5 and r_6 to classify queries for this DBMS, the obtained classification is still good. The reason for this is that we then get two classes and each of them consists of the queries executed by the same access method although the name of the access method for one class is guessed incorrectly. Hence the name of the access method in a common rule is not essential. The important thing is that the relevant queries employ the same access method.

Furthermore, even if the queries in a class employs different access methods, as long as these access methods behave similarly in terms of performance the classification is still satisfactory.

If the developer of a component DBMS is the same as the developer of the MDBS, all actual rules for choosing access methods on the DBMS are known. Some actual rules may be cost-based; i.e., if the value of a cost function is less than the value of another, an access method is chosen; otherwise, another access method is chosen. They can be used in the same way as a heuristic-based rule for classification.

In general, the more information we have about a component DBMS (rules), the better classification would result.

Many (component) DBMSs can describe (explain) the access method chosen for a (component) query. For this type of DBMSs, the common rules can be verified by running some test queries and observing their access methods. Actual rules on such a DBMS can be guessed via experiments. The common rules used for classification can be adjusted accordingly. For those DBMSs that do not provide any information about the execution of a query, the discussed common rules can be assumed to be valid. The goodness of the classification depends on the degree of agreement between the common rules and the actual rules. Since the common rules are usually robust, we expect such a classification to be acceptable. In fact, nowadays more and more DBMSs tend to provide an explain facility for execution plans (access methods). Therefore, only a few DBMSs do not provide any useful information about the execution of a query.

3.3. Classification Procedure

A query classification procedure can be described as follows. Initially, there is one class that is the entire set G of given queries. Query class G can then be divided into two smaller classes:

$$G = G_1 \cup G_2, \tag{1}$$

where $G_1 = \{unary\ queries\}$, $G_2 = \{join\ queries\}$. This classification is based on common rule r_1 and syntax information about the queries.

If we know more rules for choosing access methods for queries in a component DBMS, we can refine the classification (1). For example, if we know the common rule r_2 in Table 1 is valid for the component DBMS, we can divide G_1 into two smaller classes:

$$G_1 = G_{1\ 1} \cup G_1',$$

where

$$\begin{aligned} G_{1\ 1} = \{ & \, unary\ queries\ whose\ qualifications\ have\ at\ least\ one\ conjunct \\ & R_i.a_n = C\ where\ R_i.a_n\ is\ clustered-indexed\, \}. \end{aligned}$$

G_1' contains the queries in G_1 that are not in $G_{1\ 1}$. This classification refinement is actually based on information about query syntaxes, operand tables, and supported access methods.

If we know that the common rules $r_3 \sim r_6$ in Table 1 are also valid for the component DBMS, we can further divide G_1' into smaller classes:

$$G_1' = G_{1\ 2} \cup G_{1\ 3} \cup \cdots \cup G_{1\ 5},$$

where

$$\begin{aligned} G_{1\ 2} = \{ & \, unary\ queries\ whose\ qualifications\ have\ at\ least\ one\ conjunct \\ & R_i.a_n = C\ where\ R_i.a_n\ is\ indexed\, \} - G_{1\ 1}. \\ G_{1\ 3} = \{ & \, unary\ queries\ whose\ qualifications\ have\ at\ least\ one\ conjunct \\ & R_i.a_n\ \theta\ C\ where\ R_i.a_n\ is\ clustered-indexed\, \} - G_{1\ 1} - G_{1\ 2}, \end{aligned}$$

$$G_{1\,4} \;=\; \{\, unary\ queries\ whose\ qualifications\ have\ at\ least\ one\ conjunct$$
$$R_i.a_n\,\theta\,C\ where\ R_i.a_n\ is\ indexed\,\} \;-\; G_{1\,1} \;-\; G_{1\,2} \;-\; G_{1\,3},$$
$$G_{1\,5} \;=\; G_1 \;-\; G_{1\,1} \;-\; G_{1\,2} \;-\; G_{1\,3} \;-\; G_{1\,4}.$$

$G_{1\,1} \sim G_{1\,5}$ are the query classes generated by common rules $r_2 \sim r_6$, respectively.

Similarly, if we know that the common rules $r_7 \sim r_{10}$ in Table 2 are valid for the component DBMS, we can divide G_2 into smaller classes:

$$G_2 \;=\; G_{2\,1} \,\cup\, G_{2\,2} \,\cup\, \cdots \,\cup\, G_{2\,4},$$

where

$$G_{2\,1} \;=\; \{\, join\ queries\ whose\ qualifications\ have\ at\ least\ one\ conjunct$$
$$R_i.a_n = R_j.a_m\ where\ either\ R_i.a_n\ or\ R_j.a_m\ is\ clustered-indexed\,\},$$
$$G_{2\,2} \;=\; \{\, join\ queries\ whose\ qualifications\ have\ at\ least\ one\ conjunct$$
$$R_i.a_n = R_j.a_m\ where\ either\ R_i.a_n\ or\ R_j.a_m\ is\ indexed\,\} \;-\; G_{2\,1},$$
$$G_{2\,3} \;=\; \{\, join\ queries\ whose\ qualifications\ have\ at\ least\ one\ index-$$
$$usable\ conjunct\ for\ at\ least\ one\ operand\ table\,\} \;-\; G_{2\,1} \;-\; G_{2\,2},$$
$$G_{2\,4} \;=\; G_2 \;-\; G_{2\,1} \;-\; G_{2\,2} \;-\; G_{2\,3}.$$

$G_{2\,1} \sim G_{2\,4}$ are the query classes generated by common rules $r_7 \sim r_{10}$, respectively.

In principle, any of classes $G_{1\,i}, G_{2\,j}$ $(1 \le i \le 5; 1 \le j \le 4)$ can be further divided into smaller classes if more information is available (see Figure 2).

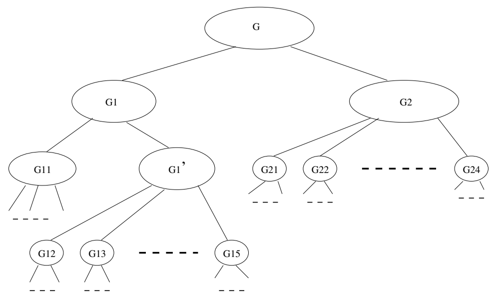

Figure 2. Classify Queries in Top-down Approach

For example, in some DBMSs, such as ORACLE 7.0, an access method by concatenating indexes may be supported to make use of an index-usable conjunct, such as $R_1.a_1 < 10 \;\vee\; R_1.a_2 = 15$ where $R_1.a_1$ and $R_1.a_2$ are indexed for a query in $G_{1\,5}$. Then the class $G_{1\,5}$ can be further divided into two smaller classes --- one $G'_{1\,5}$ contains queries having one or more index-usable conjuncts and the other $G''_{1\,5}$ contains queries without such conjuncts. In fact, index-usable conjuncts can be divided into *equality index-usable conjuncts* (i.e., $R_i.a_n = C^{(R_i.a_n)}$, where $R_i.a_n$ is clustered or non-clustered indexed, for $G_{1\,1} \cup G_{1\,2}$) and *non-equality index-usable conjuncts* (i.e., other index-usable conjuncts for $G_{1\,3} \cup G_{1\,4} \cup\ part\ of\ G'_{1\,5}$). Furthermore, equality index-usable conjuncts can be refined into *equality clustered-index-usable conjuncts* (i.e., $R_i.a_n = C^{(R_i.a_n)}$, where $R_i.a_n$ is clustered-indexed, for $G_{1\ 1}$) and *equality non-clustered-index-usable conjuncts* (i.e., $R_i.a_n = C^{(R_i.a_n)}$, where $R_i.a_n$ is non-clustered-indexed, for $G_{1\ 2}$). It is similar for non-equality index-usable conjuncts, i.e., further refining $G_{1\ 3} \cup G_{1\ 4} \cup\ part\ of\ G'_{1\ 5}$. The similar rules can be applied to refine $G_{2\ 3}$ where the situation is more complicated because left and right index-usable conjuncts may occur.

As another example, any class can be divided into smaller classes according to the (estimated) sizes of query result tables (or operand tables) because a DBMS may adopt different processing and buffering strategies to handle queries with different result sizes.

Also, the query classes can be refined according to the data type(s) of referenced column(s), because the same access method may behave differently for different data types in term of performance.

In addition, a join query class can be refined according to whether there is a joining column pair which satisfies a referential integrity constraint.

The classification of queries may vary from one DBMS or application to another. In general, a refined classification is expected to yield better cost estimates because each query class is usually more homogeneous in terms of performance. However, the overhead of maintaining the cost parameters grows as the number of query classes increases. A trade-off between estimation accuracy and maintenance overhead is required.

After a classification is given, as we will see, a cost estimation formula will be derived for each class. However, it is possible that we cannot find a satisfactory cost formula for a particular class. In that case, the classification needs to be re-considered and refined by trying to use some more information and/or guesses. Therefore, in practice, the classification procedure and the cost formula derivation may need to be iterated several times before satisfactory cost estimation formulas can be achieved.

Some component DBMSs in an MDBS may retain strong local autonomy in the sense that not much information is available at the global level for query classification. In this case, the assumed common rules and some estimated information are used to classify queries. The goodness of derived cost formulas depends on the agreement between the assumed/estimated information and the actual information in the component DBMS. Some warnings can be issued to the global query optimizer to alert this case. The global query optimizer must use such derived cost formulas with caution during query optimization.

The classification $\Re_c$ that consists of $G_{1\ 1} \sim G_{1\ 5}$ and $G_{2\ 1} \sim G_{2\ 4}$ is called a *common classification*, because it is valid for many component DBMSs (probably with some minor changes). A query class in $\Re_c$ is called a *common query class*. The common classification will be used as a representative classification in the following discussion of the query sampling method. The conclusions derived on the common classification are quite typical and can be generalized to other possible classifications.

4. Sampling Queries

Using the method discussed in Section 3, one can group queries into more homogeneous classes. It is too expensive or even impossible to perform all queries in each class to obtain cost information because the number of queries in a class is usually huge. How can we solve this problem? As we know, sampling is an example of inductive logic by which conclusions can be inferred on the basis of a limited number of instances. If each query class is considered as a population, a sample of queries can be drawn from each population. It is expected that a small number of sample queries can represent the whole population so that the cost estimation formula derived from the observed costs of sample queries can be used to give a good estimate for the cost of a query in the population. The question is how to draw such a sample of queries from a query class. This is the issue to be considered in this section.

4.1. Two-Phase Sampling Approach

There are a number of ways for a sample to be drawn from a population [4]: probability sampling (e.g., simple random sampling, stratified sampling, cluster sampling, multi-stage cluster sampling, systematic sampling, etc.), judgment sampling, and convenience sampling. Probability samples have one thing in common: the sampling units are chosen according to a probability plan. These samples usually lead to estimators whose properties can be evaluated formally. Judgment sampling is useful when the sample is to be very small, the population is very heterogeneous, or special skills are required to form a representative subset of the population. The quality of a judgment sample depends on the competence of the expert who selects the sampling units. Convenience samples are prone to bias by their very nature --- selecting sampling units that are convenient to choose almost always makes them special or different from the rest of sampling units in the population in some way.

What sampling method is suitable for our problem? One practical difficulty in sampling queries from a query class is that it is usually hard to enumerate all queries in the class because of the large number and heterogeneity of the queries. Furthermore, a sample is required to be small compared with the size of a query class, because the overhead of performing sample queries should be reasonable. Fortunately some known characteristics of queries can help us to choose sample queries and reduce the sample size. In other words, judgment sampling is useful here to choose representative sample queries for a query class based on our knowledge about the queries. However, our knowledge is usually not sufficient to get a small enough representative sample. Therefore, it is also necessary to apply other sampling methods to further reduce the size of a sample. In fact, some probability sampling methods can be applied in the second stage. Since a query class is usually very complicated, a single probability sampling technique may still not be sufficient to handle all cases well. Hence several probability sampling techniques may be combined in a sampling procedure.

In summary, we adopt a *two-phase sampling approach* to perform query sampling. The idea is to draw a sample of queries from a query class in following two phases (see Figure 3):

Phase I: use judgment sampling to select a set of representative queries, which will serve as a frame for further sampling in the second phase, from a query class based on some knowledge about the queries,

Phase II: use one or more probability sampling techniques to draw a sample of queries from the set of representative queries.

In the following two subsections, these two phases will be discussed in more details.

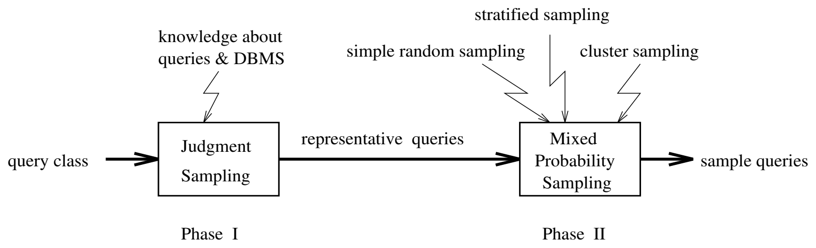

Figure 3. Two-Phase Sampling Approach

4.2. Choosing Representative Queries

What sample queries to be drawn from a query class depends on the query class. It is hard to describe query sampling without a specific query class. In this section, we mainly discuss how to draw sample queries from the common query classes $G_{1\ 1} \sim G_{1\ 5}$ and $G_{2\ 1} \sim G_{2\ 4}$ introduced in Section 3.3. Since the common classification is quite typical, many principles discovered here can be applied or extended to other possible classifications.

In this subsection, we discuss how to find queries that can represent a large number of other queries in a given query class in terms of performance behavior. Our goal is to find a set of representative queries for a given query class so that the set can serve as a good frame for further sampling for the query class. If this set of queries is used to represent the query class, many other queries can be removed from consideration in the second phase of query sampling.

We notice that the performance of two similar predicates is similar; for instance, $R_i.a_n < C$ and $R_i.a_n \leq C$; $R_i.a_n > C$ and $R_i.a_n \geq C$; $R_i.a_n \neq C$ and the '$true$' predicate. Thus they can represent each other. We, therefore, only consider the comparison operators $<,\ >,\ \neq$ and $=$ for the predicates in the representative queries for a query class. In other words, $\leq, \geq$ and nil are represented by $<,\ >$ and $\neq$, respectively.

We also notice that each query in a unary query class $G_{1\ 1} \sim G_{1\ 5}$ has a conjunct (maybe more than one) that determines the access method chosen for the query, such as $R_i.a_n = C$ (where $R_i.a_n$ is clustered-indexed) for a query in $G_{1\ 1}$. Such a key conjunct hence dominates the performance of the query. Other conjuncts, if any, in the qualification of the query usually do not affect the performance of the query as significantly as a key conjunct does.

EXAMPLE: For the following unary query $\sigma_{R_1.a_1=3\ \wedge\ R_1.a_2<10}(R_1)$ on table $R_1(a_1,\ a_2)$, where $R_1.a_1$ is indexed and $R_1.a_2$ is sequential, the key conjunct $R_1.a_1 = 3$ determines that the access method used for the query is the index scan method with a key value. Thus a component DBMS fetches all tuples that satisfy $R_1.a_1 = 3$ through the index on $R_1.a_1$. For each fetched tuple, the DBMS checks if the other conjunct $R_1.a_2 < 10$ is satisfied.

Clearly, the major cost here is to fetch the tuples from R_1. The cost for checking the second conjunct is relatively small. In other words, there is no significant performance difference between the queries with qualifications $R_1.a_1 = 3$ and $R_1.a_1 = 3 \wedge R_1.a_2 < 10$, respectively. Moreover, there is no significant performance difference among queries with qualifications $R_1.a_1 = 3$, $R_1.a_1 = 3 \wedge (R_1.a_2 < 5 \vee R_1.a_2 > 10)$, $R_1.a_1 = 3 \wedge R_1.a_2 = 9$, and other qualifications that contains key conjunct $R_1.a_1 = 3$. Hence the key conjunct should be included in the qualification of a representative query that represents all the other queries whose qualifications contain this key conjunct.

However, although key conjunct $R_1.a_1 = 3$ determines the performance of the queries with qualifications $R_1.a_1 = 3 \wedge F$, the remaining part F of such a qualification may change the size of the result table which, as we will see, is one of the input parameters for the relevant cost formula. In order to take this effect into consideration, we use simple predicates $R_1.a_m \; \omega \; C^{(1.a_m)}$ to represent F. In other words, we use the following set of qualifications

$$\{ \; R_1.a_1 = 3 \wedge R_1.a_m \, \omega \, C^{(1.a_m)} \; | \; \omega \in \{ =, \neq, <, > \} \; and \\ C^{(1.a_m)} \; is \; any \; constant \; in \; the \; domain \; of \; R_1.a_m \; and \; m = 1 \; or \; 2 \; \}$$

to represent the set of qualifications below:

$$\{ \; R_1.a_1 = 3 \wedge F \; | \; F \; is \; any \; legal \; partial \; qualification \; \}.$$

□

In general, for a unary query class $G_U \in \{G_{1\;1}, \, G_{1\;2}, \, G_{1\;3}, \, G_{1\;4}\}$, let X_P be the set of queries in G_U whose qualifications contain a key conjunct P; i.e.,

$$X_P \; = \; \{ \; \pi_{\alpha(i)}(\sigma_{P \; \wedge \; F}(R_i)) \; | \; P \wedge F \; is \; a \; legal \; qualification \; of \\ a \; query \; on \; R_i \; in \; G_U \; \}. \tag{2}$$

We use the following set of queries

$$XR_P \; = \; \{ \; \pi_{\alpha(i)}(\sigma_{P \; \wedge \; R_i.a_m \; \omega \; C^{(i.a_m)}}(R_i)) \; | \; P \wedge R_i.a_m \, \omega \, C^{(i.a_m)} \\ is \; a \; legal \; qualification \; of \; a \; query \; on \; R_i \; in \; G_U, \\ a_m \; is \; any \; column \; in \; R_i, \; and \; \omega \in \{=, \neq, <, >\} \; \} \tag{3}$$

to represent X_P. We call $R_i.a_m \; \omega \; C^{(i.a_m)}$ an *auxiliary predicate (conjunct)* of the query $\pi_{\alpha(i)}(\sigma_{P \; \wedge \; R_i.a_m \; \omega \; C^{(i.a_m)}}(R_i))$. The union of all such XR_P's for all possible key conjuncts P's is used as the set UR of representative queries for G_U; that is,

$$UR \; = \bigcup_{P \in \wp} XR_P \tag{4}$$

where $\wp$ is the set of all possible key conjuncts for the queries in G_U.

For unary query class $G_{1\;5}$, any conjunct in the qualification of a query is a key conjunct. Since there is no significant difference among the key conjuncts for such a query in terms of performance, we choose to consider the key conjuncts of simple forms[1] $R_i.a_n \; \theta \; C^{(i.a_n)}$ where $\theta \in \{<, \; >, \; =, \; \neq\}$. Using such key conjuncts in (2) $\sim$ (4), we can get the set of representative queries for $G_{1\;5}$.

Table 3 shows the possible representative query forms for the common unary query classes. The key conjunct referenced in a representative query is called a *representative key conjunct*[2].

Table 3. Representative Query Forms for Common Unary Query Classes

Query Class	Representative Query Forms	Conditions
$G_{1\ 1}$	$\pi_{\alpha^{(i)}}(\sigma_{R_i.a_n = C^{(i.a_n)} \wedge R_i.a_m\ \omega\ C^{(i.a_m)}}(R_i))$	$R_i.a_n$ is clustered-indexed
$G_{1\ 2}$	$\pi_{\alpha^{(i)}}(\sigma_{R_i.a_n = C^{(i.a_n)} \wedge R_i.a_m\ \omega\ C^{(i.a_m)}}(R_i))$	$R_i.a_n$ is indexed; ω cannot be "=" if $R_i.a_m$ is clustered-indexed
$G_{1\ 3}$	$\pi_{\alpha_1^{(i)}}(\sigma_{R_i.a_n < C_1^{(i.a_n)} \wedge R_i.a_m\ \omega_1\ C^{(i.a_m)}}(R_i))$ $\pi_{\alpha_2^{(i)}}(\sigma_{R_i.a_n > C_2^{(i.a_n)} \wedge R_i.a_k\ \omega_2\ C^{(i.a_k)}}(R_i))$	$R_i.a_n$ is clustered-indexed; ω_1 (ω_2) cannot be "=" if $R_i.a_m$ ($R_i.a_k$) is clustered-indexed or indexed
$G_{1\ 4}$	$\pi_{\alpha_1^{(i)}}(\sigma_{R_i.a_n < C_1^{(i.a_n)} \wedge R_i.a_m\ \omega_1\ C^{(i.a_m)}}(R_i))$ $\pi_{\alpha_2^{(i)}}(\sigma_{R_i.a_n > C_2^{(i.a_n)} \wedge R_i.a_k\ \omega_2\ C^{(i.a_k)}}(R_i))$	$R_i.a_n$ is indexed; ω_1 (ω_2) cannot be "=" if $R_i.a_m$ ($R_i.a_k$) is indexed; ω_1 (ω_2) can only be "$\neq$" if $R_i.a_m$ ($R_i.a_k$) is clustered-indexed
$G_{1\ 5}$	$\pi_{\alpha_1^{(i)}}(\sigma_{R_i.a_n = C_1^{(i.a_n)} \wedge R_i.a_m\ \omega_1\ C^{(i.a_m)}}(R_i))$ $\pi_{\alpha_2^{(i)}}(\sigma_{R_i.a_n \neq C_2^{(i.a_n)} \wedge R_i.a_k\ \omega_2\ C^{(i.a_k)}}(R_i))$ $\pi_{\alpha_3^{(i)}}(\sigma_{R_i.a_n < C_3^{(i.a_n)} \wedge R_i.a_p\ \omega_3\ C^{(i.a_p)}}(R_i))$ $\pi_{\alpha_4^{(i)}}(\sigma_{R_i.a_n > C_4^{(i.a_n)} \wedge R_i.a_q\ \omega_4\ C^{(i.a_q)}}(R_i))$	Only the 2nd form is allowed if $R_i.a_n$ is clustered-indexed or indexed. ω_1 (ω_2, ω_3, or ω_4) can only be "$\neq$" if $R_i.a_m$ ($R_i.a_k$, $R_i.a_p$, or $R_i.a_q$) is clustered-indexed or indexed

$\pi_{\alpha^{(i)}}(\sigma_{R_i.a_n < C^{(i.a_n)} \wedge R_i.a_m\ \omega_1\ C^{(i.a_m)}}(R_i))$, for example, is one representative query form for $G_{1\ 3}$. For each representative query form, many concrete queries can be generated by instantiating different values for the parameters of R_i, a_n, a_m, $C^{(i.a_n)}$, ω_1, $C^{(i.a_m)}$ and $\alpha^{(i)}$. For instance, the following queries

$$\pi_{R_1.a_1, R_1.a_2}(\sigma_{R_1.a_1 < 23 \wedge R_1.a_2 < 50}(R_1)),$$
$$\pi_{R_2.a_1, R_2.a_2, R_2.a_3}(\sigma_{R_2.a_1 < 4 \wedge R_2.a_3 = 122}(R_2)),$$
$$\pi_{R_1.a_4}(\sigma_{R_1.a_1 < 1 \wedge R_1.a_3 > 72}(R_1)),$$
$$\pi_{R_1.a_1, R_1.a_2, R_1.a_3, R_1.a_4}(\sigma_{R_1.a_4 < 132 \wedge R_1.a_1 \neq 91}(R_1))$$

are generated from the above representative query form, where $R_1.a_1$, $R_1.a_4$ and $R_2.a_1$ are clustered-indexed, $R_1.a_2$, $R_1.a_3$ and $R_2.a_3$ are sequential.

Let us now consider join query classes. Except the queries in $G_{2\ 3}$, each query in $G_{2\ 1}$, $G_{2\ 2}$ or $G_{2\ 4}$ has a joining predicate conjunct (maybe more than one) that determines the access method for the query, such as $R_i.a_n = R_j.a_m$ (where $R_i.a_n$ or $R_j.a_m$ is indexed) for a query in $G_{2\ 2}$, and any joining predicate $R_i.a_n = R_j.a_m$ for a query in $G_{2\ 4}$. Such a key conjunct dominates the performance of the query.

EXAMPLE: For the following join query $R_1 \underset{R_1.a_1 = R_2.a_1 \wedge R_1.a_2 = R_2.a_2}{\bowtie} R_2$, on the tables $R_1(a_1, a_2)$ and $R_2(a_1, a_2)$, where only $R_1.a_1$ is indexed and all other columns are sequential, the key conjunct $R_1.a_1 = R_2.a_1$ implies that the index join method is usually used for the query. For each tuple in R_2, a DBMS uses the index on $R_1.a_1$ to fetch the tuples satisfying $R_1.a_1 = R_2.a_1$ and then checks if the other conjunct $R_1.a_2 = R_2.a_2$ is satisfied. Hence, there is no significant difference between the two qualifications $R_1.a_1 = R_2.a_1$

and $R_1.a_1 = R_2.a_1 \ \wedge \ R_1.a_2 = R_2.a_2$ in terms of performance. However, for the above join query, if it has another conjunct $R_2.a_2 < 15$, the DBMS may use this conjunct to reduce the size of R_2 before starting the above evaluation procedure. Thus, such unary conjunct(s) may affect the performance of a join query significantly. We, therefore, use qualifications of the form $R_1.a_1 = R_2.a_1 \ \wedge \ R_1.x_1 \ \omega_1 \ C^{(1.x_1)} \ \wedge \ R_2.x_2 \ \omega_2 \ C^{(2.x_2)}$ to represent the qualifications $R_1.a_1 = R_2.a_1 \ \wedge \ \cdots$, where $x_1, \ x_2 \in \{a_1, \ a_2\}$ and $\omega_1, \ \omega_2 \in \{<, >, =, \neq\}$. The auxiliary unary conjuncts also reflect the impact of the remaining part of a qualification other than the key conjunct on the size of the result table.

□

In general, for a join query class $G_J \in \{G_{2\ 1}, \ G_{2\ 2}, \ G_{2\ 4}\}$, let Y_P be the set of queries in G_J whose qualifications contain a key conjunct P; i.e.,

$$Y_P \ = \ \{ \ \pi_{\alpha(i\ j)}(R_i \underset{P \ \wedge \ F}{\bowtie} R_j) \ \mid \ P \ \wedge \ F \ is\ a\ legal\ qualification\ of\ a\ query\ on\ R_i\ and\ R_j\ in\ G_J \ \}. \tag{5}$$

We use the following set of queries

$$\begin{aligned} YR_P \ = \ \{ \ & \pi_{\alpha(i\ j)}(R_i \underset{R_i.a_p \ \omega_1 \ C^{(i.a_p)} \ \wedge \ P \ \wedge \ R_j.a_q \ \omega_2 \ C^{(j.a_q)}}{\bowtie} R_j) \ \mid \\ & R_i.a_p \, \omega_1 \, C^{(i.a_p)} \ \wedge \ P \ \wedge \ R_j.a_q \, \omega_2 \, C^{(j.a_q)} \ is\ a\ legal\ qualification\ of \\ & a\ query\ in\ G_J, \ a_p\ is\ any\ column\ of\ R_i, \ a_q\ is\ any\ column\ of\ R_j, \\ & and\, \omega_1, \ \omega_2 \ \in \ \{<, \ >, \ =, \ \neq\} \ \} \end{aligned} \tag{6}$$

to represent Y_P. We also call $R_i.a_p \ \omega_1 \ C^{(i.a_p)}$ and $R_j.a_q \ \omega_2 \ C^{(j.a_q)}$ auxiliary predicates (conjuncts) of query $\pi_{\alpha(i\ j)}(R_i \underset{R_i.a_p \ \omega_1 \ C^{(i.a_p)} \ \wedge \ P \ \wedge \ R_j.a_q \ \omega_2 \ C^{(j.a_q)}}{\bowtie} R_j)$. The union of all such YR_P's for all possible key conjuncts P's

$$JR \ = \bigcup_{P \in \wp} \ YR_P \tag{7}$$

is used as the set of representative queries for G_J, where $\wp$ is the set of all possible key conjuncts for the queries in the class.

For a join query in class $G_{2\ 3}$, any joining predicate conjunct can be considered as a key conjunct. However, such a key conjunct P cannot determine the access method for the query alone. P plus one or more index-usable (unary) conjuncts in the qualification of a query determines the access method used for the query. We can still use the queries of forms in (6) to represent the queries in Y_P, but some restrictions need to put on the auxiliary conjuncts so that at least one of them is index-usable. In other words, we require that at least one of $R_i.a_p$ and $R_j.a_q$ is indexed or clustered-indexed and the corresponding ω_1 or ω_2 is not $\neq$.

Table 4 shows the representative query forms for the common join query classes. From the table, we can see that all the join query classes have one type of representative key conjunct; that is, $R_i.a_n = R_j.a_m$. There would be more types of representative key conjunct, e.g., $R_i.a_n < R_j.a_m$, if non-equijoin queries were considered [27].

Table 4. Representative Query Forms for Common Join Query Classes

Query Class	Representative Query Forms	Conditions
$G_{2\,1}$	$\pi_{\alpha(i\,j)}(R_i \bowtie_{R_i.a_p\,\omega_1\,C^{(i.a_p)} \wedge R_i.a_n=R_j.a_m \wedge R_j.a_q\,\omega_2\,C^{(j.a_q)}} R_j)$	$R_i.a_n$ or $R_j.a_m$ is clustered-indexed
$G_{2\,2}$	$\pi_{\alpha(i\,j)}(R_i \bowtie_{R_i.a_p\,\omega_1\,C^{(i.a_p)} \wedge R_i.a_n=R_j.a_m \wedge R_j.a_q\,\omega_2\,C^{(j.a_q)}} R_j)$	$R_i.a_n$ or $R_j.a_m$ is indexed; neither $R_i.a_n$ nor $R_j.a_m$ is clustered-indexed
$G_{2\,3}$	$\pi_{\alpha(i\,j)}(R_i \bowtie_{R_i.a_p\,\omega_1\,C^{(i.a_p)} \wedge R_i.a_n=R_j.a_m \wedge R_j.a_q\,\omega_2\,C^{(j.a_q)}} R_j)$	$R_i.a_n$ and $R_j.a_m$ are sequential; either $R_i.a_p$ or $R_j.a_q$ is clustered-indexed or indexed, and the corresponding ω_l is not $\neq$ ($l = 1$ or 2)
$G_{2\,4}$	$\pi_{\alpha(i\,j)}(R_i \bowtie_{R_i.a_p\,\omega_1\,C^{(i.a_p)} \wedge R_i.a_n=R_j.a_m \wedge R_j.a_q\,\omega_2\,C^{(j.a_q)}} R_j)$	$R_i.a_n$ and $R_j.a_m$ are sequential; ω_1 (ω_2) can only be "$\neq$" if $R_i.a_p$ ($R_j.a_q$) is clustered-indexed or indexed

4.3. *Selecting Random Sample Queries*

A query class is greatly reduced to a set of representative queries. However, as mentioned, the set of representative queries is usually still quite large. Hence, we use the set of representative queries as a frame and apply several probability sampling methods to draw a random sample of queries from the frame for the query class. We use the final (small) sample of queries to represent the query class.

By using too small a sample, however, poor estimates of cost formula (regression) coefficients may result --- leading to poor estimates of query costs. Thus, there is a minimum sample size requirement. A commonly used rule [17] is to sample at least $10 * (n + 1)$ observations for a regression formula with n coefficients. How to apply probability sampling methods to get a sample with a desirable size is the issue to be discussed in this subsection.

4.3.1. Random Unary Sample Queries Let us first consider how to draw a sample from a unary query class.

For a given unary query class G_U, any column that can be referenced in the key conjunct of a representative query for G_U is called an *eligible column*. For a component database DB, let Δ_U be the set of all eligible columns in the tables in DB for G_U. For example, Δ_U for common query class $G_{1\,1}$ consists of all clustered-indexed columns in the tables in DB. If DB consists of tables $R_1,\ R_2,\ \cdots,\ R_K$, clearly,

$$\Delta_U \;=\; \Delta_{U\;R_1} \;\cup\; \Delta_{U\;R_2} \;\cup\; \cdots \;\cup\; \Delta_{U\;R_K} \;\;, \tag{8}$$

where $\Delta_{U\ R_i}$ $(1 \leq i \leq K)$ is the set of eligible columns in table R_i. We assume that G_U is not empty. Then $|\Delta_U| \neq 0$ since a non-empty G_U should have at least one eligible column for its queries.

Let H_U denote the set of representative queries for G_U, which is now the frame for further sampling. To draw a sample query from H_U, we need to determine three parts of the query: (1) key conjunct, (2) auxiliary conjunct, and (3) project list.

Let us first consider a G_U where every eligible column induces the same number of types of representative key conjuncts. For example, if G_U is $G_{1\ 1}$ or $G_{1\ 2}$, every $R_i.a_n$ induces one type of representative key conjuncts, namely, $R_i.a_n = C^{(i.a_n)}$; if G_U is $G_{1\ 3}$ or $G_{1\ 4}$, every $R_i.a_n$ induces two types of representative key conjuncts, namely, $R_i.a_n < C_1^{(i.a_n)}$ and $R_i.a_n > C_2^{(i.a_n)}$ (see Table 3). However, for $G_U = G_{1\ 5}$, every eligible column that is sequential induces four types of representative key conjuncts, while every eligible column that is clustered-indexed or indexed induces only one type of representative key conjuncts. The situation when $G_U = G_{1\ 5}$ will be considered later on.

A component database normally does not contain a very large number of tables. Hence an appropriate sampling principle is to choose at least one sample query on each table that has at least one eligible column. These sample queries would give us information about the underlying component DBMS for performing queries on all tables, while the size of the sample is still not large.

Let M_U denote the minimum sample size that is required for G_U. We need to draw a sample with a size greater than or equal to M_U. But we do not like the sample size to be much greater than M_U because executing extra sample queries requires extra cost. This is another principle to be used when we draw sample queries.

Let b_U (> 0) be the number of types of representative key conjuncts that can be induced by each eligible column. Let $\widetilde{X}$ denote a value randomly chosen from all possible values of parameter X. For example, $\widetilde{C}^{(i.a_n)}$, $R_i.\widetilde{x}$, $\widetilde{\alpha}^{(i)}$, and $\widetilde{\omega} \in \{<, >, =, \neq\}$ represent a constant randomly chosen from the domain of $R_i.a_n$, a column randomly chosen from all columns of R_i, a project (column) list randomly chosen from all non-empty subsets of columns of R_i, and an operator randomly chosen from $\{<, >, =, \neq\}$, respectively.

To draw a sample of queries from G_U, we consider the following cases.

Case 1: $|\Delta_U| * b_U = M_U$**.** In this case, we use all eligible columns in Δ_U to draw sample queries. For each eligible column, we draw b_U sample queries, i.e., one for each representative key conjunct type. Hence we get exactly $|\Delta_U| * b_U = M_U$ queries in a sample.

More specifically, for a given eligible column $R_i.a_n \in \Delta_U$, each allowed comparison operator θ, e.g., $<$, $>$ or $=$, corresponds to one type of representative key conjunct $R_i.a_n\ \theta\ C^{(i.a_n)}$, where $C^{(i.a_n)}$ is a parameter. The key conjunct of a random sample query corresponding to $R_i.a_n\ \theta\ C^{(i.a_n)}$ is obtained by instantiating $C^{(i.a_n)}$ by a constant randomly chosen from the domain of $R_i.a_n$, i.e., $\widetilde{C}^{(i.a_n)}$.

The auxiliary predicate (conjunct) $R_i.a_m\ \omega\ C^{(i.a_m)}$ in a sample query is randomly chosen from all alternatives. That is, the referenced column $R_i.a_m$ is randomly chosen from all allowed columns, i.e., $R_i.\widetilde{x}$; the comparison operator ω is randomly chosen from all allowed operators, i.e., $\widetilde{\omega}$; the constant $C^{(i.a_m)}$ is randomly chosen from the domain of the corresponding column, i.e., $\widetilde{C}^{(i.\widetilde{x})}$.

The project list $\alpha^{(i)}$ for a sample query is randomly chosen from all non-empty subsets of columns in the operand table R_i, i.e., $\widetilde{\alpha}^{(i)}$.

For example, if $R_1.a_1$ is a clustered-indexed column, the two sample queries induced by $R_1.a_1$ for query class $G_{1\ 3}$ are

$$\pi_{\widetilde{\alpha}_1^{(1)}}(\sigma_{R_1.a_1<\widetilde{C}_1^{(1.a_1)} \wedge R_1.\widetilde{x}_1\ \widetilde{\omega}_1\ \widetilde{C}^{(1.\widetilde{x}_1)}}(R_1)) \ ,$$
$$\pi_{\widetilde{\alpha}_2^{(1)}}(\sigma_{R_1.a_1>\widetilde{C}_2^{(1.a_1)} \wedge R_1.\widetilde{x}_2\ \widetilde{\omega}_2\ \widetilde{C}^{(1.\widetilde{x}_2)}}(R_1)) \ .$$

Note that $\widetilde{\omega}_1$ ($\widetilde{\omega}_2$) is randomly chosen from $\{<,\ >,\ \neq\}$ if $R_1.\widetilde{x}_1$ ($R_1.\widetilde{x}_2$) is clustered-indexed, and randomly chosen from $\{<,\ >,\ =,\ \neq\}$ otherwise. Subscripts are used here to distinguish random values chosen at different times.

In fact, the sampling method used in this case is a stratified multi-stage cluster sampling method that combines stratified sampling with multi-stage cluster sampling.

First, the frame H_U is separated into strata, where each stratum consists of the representative queries whose representative key conjuncts are of the same type and the referenced column is instantiated by one eligible column. For example, for $G_{1\ 3}$, if there are two tables $R_1(a_1, a_2)$ and $R_2(a_1, a_2, a_3)$ in DB, where $R_1.a_1$ and $R_2.a_1$ are clustered-indexed and others are sequential, then H_U are stratified into $\{\ STRA_1,\ STRA_2,\ STRA_3,\ STRA_4\ \}$ where

$$\begin{aligned}
STRA_1 \;=\; \{\; & \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1<C^{(1.a_1)} \wedge R_1.a_m\ \omega\ C^{(1.a_m)}}(R_1)) \;\mid\; R_1.a_1\ is\ fixed, \\
& but\ C^{(1.a_1)},\ R_1.a_m,\ \omega,\ C^{(1.a_m)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ , \\
STRA_2 \;=\; \{\; & \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1>C^{(1.a_1)} \wedge R_1.a_m\ \omega\ C^{(1.a_m)}}(R_1)) \;\mid\; R_1.a_1\ is\ fixed, \\
& but\ C^{(1.a_1)},\ R_1.a_m,\ \omega,\ C^{(1.a_m)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ , \\
STRA_3 \;=\; \{\; & \pi_{\alpha^{(2)}}(\sigma_{R_2.a_1<C^{(2.a_1)} \wedge R_2.a_m\ \omega\ C^{(2.a_m)}}(R_2)) \;\mid\; R_2.a_1\ is\ fixed, \\
& but\ C^{(2.a_1)},\ R_2.a_m,\ \omega,\ C^{(2.a_m)},\ \alpha^{(2)}\ can\ be\ any\ valid\ values\ \}\ , \\
STRA_4 \;=\; \{\; & \pi_{\alpha^{(2)}}(\sigma_{R_2.a_1>C^{(2.a_1)} \wedge R_2.a_m\ \omega\ C^{(2.a_m)}}(R_2)) \;\mid\; R_2.a_1\ is\ fixed, \\
& but\ C^{(2.a_1)},\ R_2.a_m,\ \omega,\ C^{(2.a_m)},\ \alpha^{(2)}\ can\ be\ any\ valid\ values\ \}\ .
\end{aligned}$$

Secondly, each stratum is separated into clusters, then into sub-clusters, $\cdots$, and so on, by instantiating different values for one parameter, then for another, and so on. For example, the above $STRA_1$ is first separated into the following clusters by instantiating different values for constant parameter $C^{(1.a_1)}$:

$$\begin{aligned}
CLUS_1 \;=\; \{\; & \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1<C_1^{(1.a_1)} \wedge R_1.a_m\ \omega\ C^{(1.a_m)}}(R_1)) \;\mid\; R_1.a_1,\ C_1^{(1.a_1)} \\
& are\ fixed,\ but\ R_1.a_m,\ \omega,\ C^{(1.a_m)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ , \\
CLUS_2 \;=\; \{\; & \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1<C_2^{(1.a_1)} \wedge R_1.a_m\ \omega\ C^{(1.a_m)}}(R_1)) \;\mid\; R_1.a_1,\ C_2^{(1.a_1)} \\
& are\ fixed,\ but\ R_1.a_m,\ \omega,\ C^{(1.a_m)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ , \\
CLUS_3 \;=\; \{\; & \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1<C_3^{(1.a_1)} \wedge R_1.a_m\ \omega\ C^{(1.a_m)}}(R_1)) \;\mid\; R_1.a_1,\ C_3^{(1.a_1)}
\end{aligned}$$

$$are\ fixed,\ but\ R_1.a_m,\ \omega,\ C^{(1.a_m)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ ,$$

......

The cluster $CLUS_1$, for instance, is then divided into the following sub-clusters by instantiating different valid columns for $R_1.a_m$:

$$\begin{aligned} CLUS_{1\ 1} &= \{\ \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1 < C_1^{(1.a_1)}\ \wedge\ R_1.a_1\ \omega\ C^{(1.a_1)}}(R_1))\ \mid\ R_1.a_1,\ C_1^{(1.a_1)}, \\ &\quad R_1.a_1\ are\ fixed,\ but\ \omega,\ C^{(1.a_1)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ , \\ CLUS_{1\ 2} &= \{\ \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1 < C_1^{(1.a_1)}\ \wedge\ R_1.a_2\ \omega\ C^{(1.a_2)}}(R_1))\ \mid\ R_1.a_1,\ C_1^{(1.a_1)}, \\ &\quad R_1.a_2\ are\ fixed,\ but\ \omega,\ C^{(1.a_2)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ . \end{aligned}$$

Each sub-cluster can be further divided into smaller and smaller clusters by instantiating different values for ω, $C^{(1.a_m)}$ $(m = 1$ or $2)$ and $\alpha^{(1)}$, one after another.

Multi-stage cluster sampling is used to draw sample queries from each stratum. More precisely, a simple random sample of size one is drawn from a set of clusters at each level. The final sample SP_U consists of all the queries resulting in the end.

For example, the simple random sample drawn from $\{\ CLUS_1, CLUS_2, CLUS_3, \cdots\ \}$ is $\{\ CLUS_{\tilde{i}}\ \}$, where[3]

$$\begin{aligned} CLUS_{\tilde{i}} &= \{\ \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1 < \widetilde{C}^{(1.a_1)}\ \wedge\ R_1.a_m\ \omega\ C^{(1.a_m)}}(R_1))\ \mid\ R_1.a_1,\ \widetilde{C}^{(1.a_1)} \\ &\quad are\ fixed,\ but\ R_1.a_m,\ \omega,\ C^{(1.a_m)},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ . \end{aligned}$$

The simple random sample drawn from $\{\ CLUS_{\tilde{i}\,1},\ CLUS_{\tilde{i}\,2}\ \}$ is $\{\ CLUS_{\tilde{i}\,\tilde{j}}\ \}$, where

$$\begin{aligned} \{\ CLUS_{\tilde{i}\,\tilde{j}}\ \} &= \{\ \pi_{\alpha^{(1)}}(\sigma_{R_1.a_1 < \widetilde{C}_1^{(1.a_1)}\ \wedge\ R_1.\tilde{x}\ \omega\ C^{(1.\tilde{x})}}(R_1))\ \mid\ R_1.a_1,\ \tilde{\alpha}^{(1)}, \\ &\quad R_1.\tilde{x}\ are\ fixed,\ but\ \omega,\ C^{(1.\tilde{x})},\ \alpha^{(1)}\ can\ be\ any\ valid\ values\ \}\ . \end{aligned}$$

After a simple random value is chosen for each of remaining parameters ω, $C^{(1.\tilde{x})}$ and $\alpha^{(1)}$, we obtain a concrete query:

$$\pi_{\tilde{\alpha}^{(1)}}(\sigma_{R_1.a_1 < \widetilde{C}^{(1.a_1)}\ \wedge\ R_1.\tilde{x}\ \tilde{\omega}\ \widetilde{C}^{(1.\tilde{x})}}(R_1))\ ,$$

which is the sample query that we draw from $STRA_1$. Similarly, a sample query can be drawn from each of $STRA_2 \sim STRA_4$. If we use subscripts to distinguish different random values drawn at different times from a set of allowed values, the sample of queries drawn from $G_{1\ 3}$ in the above example can be described as follows:

$$\begin{aligned} \{\ &\pi_{\tilde{\alpha}_1^{(1)}}(\sigma_{R_1.a_1 < \widetilde{C}_1^{(1.a_1)}\ \wedge\ R_1.\tilde{x}_1\ \tilde{\omega}_1\ \widetilde{C}^{(1.\tilde{x}_1)}}(R_1))\ , \\ &\pi_{\tilde{\alpha}_2^{(1)}}(\sigma_{R_1.a_1 > \widetilde{C}_2^{(1.a_1)}\ \wedge\ R_1.\tilde{x}_2\ \tilde{\omega}_2\ \widetilde{C}^{(1.\tilde{x}_2)}}(R_1))\ , \\ &\pi_{\tilde{\alpha}_1^{(2)}}(\sigma_{R_2.a_1 < \widetilde{C}_1^{(2.a_1)}\ \wedge\ R_2.\tilde{y}_1\ \tilde{\omega}_3\ \widetilde{C}^{(2.\tilde{y}_1)}}(R_2))\ , \\ &\pi_{\tilde{\alpha}_2^{(2)}}(\sigma_{R_2.a_1 > \widetilde{C}_2^{(2.a_1)}\ \wedge\ R_2.\tilde{y}_2\ \tilde{\omega}_4\ \widetilde{C}^{(2.\tilde{y}_2)}}(R_2))\quad \}\ . \end{aligned}$$

Case 2: $|\Delta_U| * b_U > M_U$. In this case, if we still apply the sampling procedure in Case 1, the resulting sample size $|\Delta_U| * b_U$ may be much larger than necessary (M_U). In fact, it is sufficient to consider only part of all eligible columns during query sampling. Which part should we consider? If we draw a simple random sample of eligible columns from Δ_U, or we consider each table as a cluster and perform cluster sampling, some tables may not be considered. In other words, cost information obtained in such a way may not reflect performance of queries on all tables well.

In order to guarantee that there is at least one sample query on each table in the underlying component database while the sample size is still kept not much greater than M_U, we employ a sampling procedure described below.

The set H_U of representative queries is stratified such that all the queries whose key conjuncts reference eligible columns in the same $\Delta_{U\ R_i}$ comprise a stratum; i.e.,

$$H_U \ = \ H_{U\ R_1} \ \cup \ H_{U\ R_2} \ \cup \ \cdots \ H_{U\ R_K} \ \ ,$$

where $H_{U\ R_i}$ is the stratum corresponding to table R_i. A certain percent $\eta\%$ of eligible columns in each $\Delta_{U\ R_i}$ are selected for sample queries. In other words, a simple random sample $\Delta_{U\ R_i}^{(\eta\%)}$ of size $\lceil |\Delta_{U\ R_i}| * \eta\% \rceil$ is drawn from $\Delta_{U\ R_i}$. Let $H_{U\ R_i}^{(\eta\%)}$ be the set of representative queries associated with $\Delta_{U\ R_i}^{(\eta\%)}$. It is drawn from the stratum $H_{U\ R_i}$. Let

$$H_U^{(\eta\%)} = H_{U\ R_1}^{(\eta\%)} \ \cup \ H_{U\ R_2}^{(\eta\%)} \ \cup \ \cdots \ \cup \ H_{U\ R_K}^{(\eta\%)} \ \ ,$$

and

$$\Delta_U^{(\eta\%)} = \Delta_{U\ R_1}^{(\eta\%)} \ \cup \ \Delta_{U\ R_2}^{(\eta\%)} \ \cup \ \cdots \ \cup \ \Delta_{U\ R_K}^{(\eta\%)} \ \ .$$

We, then, apply the same stratified multi-stage cluster sampling procedure described in Case 1 to obtain a sample SP_U of queries from H_U by considering $H_U^{(\eta\%)}$ as H_U and $\Delta_U^{(\eta\%)}$ as Δ_U in Case 1.

Now the question is what percentage of eligible columns we should select from each $\Delta_{U\ R_i}$ so that the size of the final sample SP_U is greater than or equal to the required minimum size M_U.

THEOREM 1 *For* $|\Delta_U| * b_U > M_U$, *if*

$$\eta = 100 * M_U / (|\Delta_U| * b_U), \tag{9}$$

then $M_U \leq |SP_U| \leq |\Delta_U| * b_U$.

Proof: From (8) and the fact that $\Delta_{U\ R_i} \cap \Delta_{U\ R_j} = \emptyset$ for $i \neq j$, we have

$$|\Delta_U| \ = \ |\Delta_{U\ R_1}| + |\Delta_{U\ R_2}| + \cdots + |\Delta_{U\ R_K}| \ \ . \tag{10}$$

Consider the total number of sample queries

$$\begin{aligned} |SP_U| \ &= \ \sum_{i=1}^{K} b_U * \lceil |\Delta_{U\ R_i}| * \eta\% \rceil \\ &= \ \sum_{i=1}^{K} b_U * \lceil |\Delta_{U\ R_i}| * 100 * M_U / (b_U * |\Delta_U| * 100) \rceil \end{aligned}$$

$$\begin{aligned} &= \sum_{i=1}^{K} b_U * \lceil |\Delta_{U\ R_i}| * M_U/(b_U * |\Delta_U|) \rceil \qquad (11) \\ &\geq \sum_{i=1}^{K} b_U * |\Delta_{U\ R_i}| * M_U/(b_U * |\Delta_U|) \\ &= \sum_{i=1}^{K} |\Delta_{U\ R_i}| * M_U/|\Delta_U| \\ &= \frac{M_U}{|\Delta_U|} \sum_{i=1}^{K} |\Delta_{U\ R_i}| \ = \ M_U. \end{aligned}$$

From (11) and $M_U/(b_U * |\Delta_U|) \ < \ 1$, we have

$$|SP_U| \ \leq \ \sum_{i=1}^{K} b_U * \lceil |\Delta_{U\ R_i}| \rceil \ = \ b_U * \sum_{i=1}^{K} |\Delta_{U\ R_i}| \ = \ b_U * |\Delta_U|. \qquad \blacksquare$$

Theorem 1 states that if we choose η as in (9), the size of a sample obtained by the above sampling procedure satisfies the minimum sample size requirement and is not greater than $|\Delta_U| * b_U$. The sample size is usually smaller than $|\Delta_U| * b_U$ although it equals to the latter in the worst case.

In fact, Case 1 can be considered as a special case of Case 2 when $\eta\% = 100\%$.

Case 3: $|\Delta_U| * b_U < M_U$**.** In this case, it is not sufficient to consider all eligible columns and choose b_U sample queries for each eligible column. Clearly, all eligible columns need to be considered, but more than b_U sample queries need to be chosen for each eligible column in order to have the sample size satisfy the minimum sample size requirement.

The sampling method to be used in this case is the same as the one in Case 1. However, we draw more than one cluster during the first stage of the multi-stage cluster sampling. We achieve this by instantiating more than one constant for the constant parameter in each representative key conjunct type. For simplicity, we draw the same number of clusters from each stratum. As for other parameters in a representative query form, such as the column parameter in an auxiliary predicate and the project list parameter, we still instantiate one random value.

How many clusters need to be drawn in the first stage so that the final sample SP_U of queries satisfies the minimum sample size requirement? Let λ_U be the number of clusters needed to be drawn from each stratum. We have

THEOREM 2 *For* $|\Delta_U| * b_U < M_U$, *if*

$$\lambda_U = \lceil M_U/(|\Delta_U| * b_U) \rceil, \qquad (12)$$

then $M_U \leq |SP_U| < (M_U + |\Delta_U| * b_U)$.

Proof:

$$|SP_U| \ = \ \lambda_U * b_U * \sum_{i=1}^{K} |\Delta_{U\ R_i}| \ = \ \lambda_U * b_U * |\Delta_U|$$

$$\begin{aligned} &= \lceil M_U/(|\Delta_U| * b_U) \rceil * b_U * |\Delta_U| \\ &\geq M_U/(|\Delta_U| * b_U) * b_U * |\Delta_U| \;=\; M_U. \end{aligned} \tag{13}$$

From (13) and $\lceil M_U/(|\Delta_U| * b_U)\rceil < M_U/(|\Delta_U| * b_U) + 1$, we have

$$|SP_U| < (M_U/(|\Delta_U| * b_U) + 1) * b_U * |\Delta_U| = M_U + b_U * |\Delta_U|.$$

■

Theorem 2 states that a sample produced by the sampling procedure satisfies the minimum sample size requirement and does not include more than $b_U * |\Delta_U| - 1$ extra sample queries.

In fact, all three cases discussed above can be unified as follows. Case 1 can be considered as $\eta = 100$ and $\lambda_U = 1$. Case 2 can be considered as $\lambda_U = 1$ and η to be determined by Theorem 1. Case 3 can be considered as $\eta = 100$ and λ_U to be determined by Theorem 2. Note that the formula (12) for λ_U actually works for all three cases.

Now let us consider how to draw sample queries from $G_{1\ 5}$ which is in the situation where different eligible columns may induce different numbers of representative key conjunct types.

For $G_{1\ 5}$, the set Δ_U of eligible columns consists of all possible columns. However, an (clustered or non-clustered) indexed column and a sequential column induce different numbers of representative key conjunct types. Let $\Delta_U = \Delta'_U \cup \Delta''_U$, where Δ'_U consists of all (clustered or non-clustered) indexed columns, and Δ''_U consists of all sequential columns. Without loss of generality, assume $\Delta'_U \neq \emptyset$ and $\Delta''_U \neq \emptyset$. Let H'_U and H''_U be the sets of representative queries associated with Δ'_U and Δ''_U, respectively. For a given minimum size requirement M_U, we can choose M'_U and M''_U such that $M_U = M'_U + M''_U$. For instance, let[4] $M'_U = \lfloor M_U * |\Delta'_U|/(|\Delta'_U| + 4 * |\Delta''_U|)\rfloor$ and $M''_U = M_U - M'_U$. The previous sampling procedures can then be applied to draw a sample SP'_U (SP''_U) from H'_U (H''_U) by considering Δ'_U (Δ''_U) as Δ_U and M'_U (M''_U) as M_U. The final sample SP_U is the union of SP'_U and SP''_U.

The sampling procedure for $G_{1\ 5}$ actually stratifies H_U into H'_U and H''_U first, then applies the previous sampling procedures to each of them.

From the above discussion, we can see that a mixture of simple random sampling, stratified sampling and cluster sampling is, in fact, used to sample queries from a unary query class for all cases.

4.3.2. Random Join Sample Queries Drawing sample queries from a join query class is more complicated than drawing sample queries from a unary query class. One principle used in Section 4.3.1 for a unary query class is that at least one sample query is drawn for each possible operand, i.e., a table that has at least one eligible column. However, the principle may not be good for sampling queries from a join query class because the number of possible operands (joining table pairs) is usually large. Even if we draw only one sample query for each pair of joining tables, the sample size may still be much larger than the required minimum sample size, which may not be good because performing a join sample query is usually quite expensive. It is, therefore, desired to draw a sample of join queries with a size not only greater than but also close to the required minimum sample size.

Note that, in practice, not every pair of columns can be a joining column pair referenced in a joining predicate. Such a pair of columns must be comparable by ''$=$''. For simplicity, we assume that all pairs of columns are comparable in the following discussion.

If two columns $R_i.a_n$ and $R_j.a_m$ can be referenced by the representative key conjunct, i.e., a joining predicate, of a representative query for a join query class, the pair $(R_i.a_n,\ R_j.a_m)$ is called an *eligible joining column pair* for the query class. For example, $(R_1.a_1,\ R_2.a_1)$ is an eligible joining column pair for query class $G_{2\ 1}$ if either $R_1.a_1$ or $R_2.a_1$ is clustered-indexed.

In a representative query, there are two auxiliary conjuncts (predicates) for the left and right joining tables, respectively. For join query class $G_{2\ 3}$, at least one referenced column in an auxiliary conjunct for a representative query is index-usable. For other join query classes, any column can be referenced in an auxiliary conjunct for a representative query. If two columns $R_i.a_p$ and $R_j.a_q$ can be referenced by the left and right auxiliary conjuncts of a representative query for a join query class, respectively, the pair $(R_i.a_p,\ R_j.a_q)$ is called an *eligible auxiliary column pair* for the query class.

Let Δ_J be the set of all eligible joining column pairs for a given join query class G_J in a component database. Let Δ_{JA} be the set of all eligible auxiliary column pairs for G_J, and $\Delta_{JA}|_{(R_i,R_j)}$ is the subset of Δ_{JA} with the restriction that the first column in a pair is in R_i and the second column in the pair is in R_j. Let H_J be the set of representative queries for G_J, which serves as the frame for the second phase sampling. We assume that G_J is not empty. Then Δ_J and H_J are not empty either because a non-empty G_J has at least one eligible joining column pair and one representative query. Let M_J be the minimum sample size required for G_J and $\widetilde{\alpha}^{(i\ j)}$ be a project list randomly chosen from all non-empty subsets of columns of R_i and R_j.

Note that each eligible joining column pair $(R_i.a_n,\ R_j.a_m)$ induces only one representative key conjunct type, i.e., $R_i.a_n = R_j.a_m$, in our case. However, an eligible joining column pair might induce more than one representative key conjunct types if non-equijoin queries were considered. To make our results more general, in the following discussion, we still include a parameter $b_J\ (> 0)$, like b_U for a unary query class, to denote the number of representative key conjunct types induced by an eligible joining column pair for G_J. However, as just mentioned, $b_J \equiv 1$ in our case. The situation where different eligible joining column pairs for a join query class may induce different numbers of representative key conjunct types can be handled in a similar way used for a unary query class.

To draw a sample from a join query class, let us consider the following cases:

Case 1: $|\Delta_J| * b_J \geq M_J$. In this case, we draw a simple random sample $\widetilde{\Delta}_J^{(1)}$ of eligible joining column pairs with size $\lceil M_J/b_J \rceil$ from Δ_J. For each eligible joining column pair in $\widetilde{\Delta}_J^{(1)}$, we draw b_J sample queries, i.e., one for each representative key conjunct type. For a join sample query on R_i and R_j, the auxiliary column pair is randomly chosen from $\Delta_{JA}|_{(R_i,R_j)}$; the comparison operators and constants in the auxiliary predicates (conjuncts) are randomly chosen from the relevant domains; the project list is randomly chosen from the non-empty subsets of the union of columns of the two joining tables.

For example, if $(R_1.a_1,\ R_2.a_1)$ is an eligible joining column pair for join query class $G_{2\ 1}$, the sample query associated with $(R_1.a_1,\ R_2.a_1)$ that we draw from $G_{2\ 1}$ is

$$\pi_{\widetilde{\alpha}^{(1\ 2)}}\left(R_1 \underset{R_1.x_1\ \widetilde{\omega}_1\ \widetilde{C}^{(1.x_1)}\ \wedge\ R_1.a_1=R_2.a_1\ \wedge\ R_2.x_2\ \widetilde{\omega}_2\ \widetilde{C}^{(2.x_2)}}{\bowtie} R_2\right),$$

where $(R_1.x_1,\ R_2.x_2)$ is randomly chosen from $\Delta_{JA}|_{(R_1, R_2)}$; $\widetilde{\omega}_1$ and $\widetilde{\omega}_2$ are randomly chosen from $\{<, >, =, \neq\}$; $\widetilde{C}^{(1.x_1)}$ and $\widetilde{C}^{(2.x_2)}$ are randomly chosen from the domains of $R_1.x_1$ and $R_2.x_2$, respectively; $\widetilde{\alpha}^{(1\ 2)}$ is randomly chosen from all the columns of R_1 and R_2.

The size of such a sample SP_J is $b_J * \lceil M_J/b_J \rceil$. For a general $b_J > 0$, we have

THEOREM 3 *Let SP_J be the set of sample queries drawn from H_J as described above. Then $M_J \leq |SP_J| \leq M_J + (b_J - r)$, where r $(0 \leq r < b_J)$ is the remainder of the division M_J by b_J.*

Proof: We have

$$|SP_J| \ = \ b_J * \lceil M_J/b_J \rceil \ \geq \ b_J * M_J/b_J \ = \ M_J. \tag{14}$$

Let $M_J/b_J = k + r/b_J$ where k is the quotient and r is the remainder. Hence

$$\begin{aligned} |SP_J| \ &= \ b_J * \lceil k + r/b_J \rceil \ \leq \ b_J * (k+1) \\ &= \ b_J * k + b_J \ = \ M_J + (b_J - r). \end{aligned}$$

■

Theorem 3 claims that the above sampling procedure draws a sample with a size satisfying the minimum size requirement and with at most $b_J - r \leq b_J$ extra sample queries. Since b_J is usually quite small, the drawn sample has the desirable property that the size is very close to the required minimum size. In fact, for our case $b_J \equiv 1$, $|SP_J| = b_J * \lceil M_J/b_J \rceil = M_J$.

Case 2: $|\Delta_J| * b_J < M_J$**.** Clearly, in this case, all eligible joining column pairs need to be used. However, drawing only b_J sample queries for each eligible pair is not sufficient.

If we insisted on drawing the same number of sample queries for all eligible joining column pairs, like Case 3 for a unary query class, we could have up to $b_J * |\Delta_J| - 1$ extra sample queries (like Theorem 2). This number of extra sample queries is reasonable for a unary query class, but it may not be acceptable for a join query class, because a join query is often much more expensive to execute.

To improve the sampling procedure, we choose a random subset $\widetilde{\Delta}_J^{(2)}$ of Δ_J with size $\lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil$, where $\lambda_J = \lfloor M_J/(|\Delta_J| * b_J) \rfloor$. The following theorem shows that the chosen size of $\widetilde{\Delta}_J^{(2)}$ is feasible although $\widetilde{\Delta}_J^{(2)}$ may be an empty or whole set.

THEOREM 4 *If $\lambda_J = \lfloor M_J/(b_J * |\Delta_J|) \rfloor$, then $0 \leq |\widetilde{\Delta}_J^{(2)}| = \lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil \leq |\Delta_J|$.*

Proof:

$$\begin{aligned}|\widetilde{\Delta}_J^{(2)}| &= \lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil \;\geq\; (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \\ &= (M_J - \lfloor M_J/(b_J * |\Delta_J|) \rfloor * b_J * |\Delta_J|)/b_J \\ &\geq (M_J - b_J * |\Delta_J| * M_J/(b_J * |\Delta_J|))/b_J \;=\; 0.\end{aligned}$$

Notice that

$$\begin{aligned}M_J - \lambda_J * b_J * |\Delta_J| &= M_J - \lfloor M_J/(b_J * |\Delta_J|) \rfloor * b_J * |\Delta_J| \\ &< M_J - (M_J/(b_J * |\Delta_J|) - 1) * b_J * |\Delta_J| \;=\; b_J * |\Delta_J|.\end{aligned}$$

Hence $(M_J - \lambda_J * b_J * |\Delta_J|)/b_J \;<\; |\Delta_J|$. Therefore, $\lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil \;\leq\; |\Delta_J|$. ■

For each eligible joining column pair in $\widetilde{\Delta}_J^{(2)}$, we draw $(\lambda_J + 1) * b_J$ sample queries from H_J; namely, using $(\lambda_J + 1)$ sets of randomly-chosen left and right auxiliary predicates and project lists to form $(\lambda_J + 1)$ sample queries for each representative key conjunct type associated with the given joining column pair. For each eligible joining column pair in $\Delta_J - \widetilde{\Delta}_J^{(2)}$, we draw $\lambda_J * b_J$ sample queries from H_J; namely, using λ_J sets of randomly-chosen left and right auxiliary predicates and project lists to form λ_J sample queries for each representative key conjunct type that is associated with the given joining column pair. If $\widetilde{\Delta}_J^{(2)}$ is a non-empty proper subset of Δ_J, some eligible joining column pairs in Δ_J generate $(\lambda_J + 1) * b_J$ sample queries, while others generate $\lambda_J * b_J$ sample queries. If $\widetilde{\Delta}_J^{(2)}$ is an empty or whole set, each eligible joining column pair generate $\lambda_J * b_J$ or $(\lambda_J + 1) * b_J$ sample queries, respectively.

THEOREM 5 *Let SP_J be the set of sample queries drawn from H_J as described above. Then $|SP_J| = b_J * (|\widetilde{\Delta}_J^{(2)}| + \lambda_J * |\Delta_J|)$, and $M_J \leq |SP_J| \leq M_J + b_J$.*

Proof: From the above discussion, we have

$$\begin{aligned}|SP_J| &= (\lambda_J + 1) * b_J * |\widetilde{\Delta}_J^{(2)}| + \lambda_J * b_J * (|\Delta_J| - |\widetilde{\Delta}_J^{(2)}|) \\ &= b_J * (|\widetilde{\Delta}_J^{(2)}| + \lambda_J * |\Delta_J|) \;.\end{aligned}$$

Then

$$\begin{aligned}|SP_J| &= b_J * (|\widetilde{\Delta}_J^{(2)}| + \lambda_J * |\Delta_J|) \\ &= b_J * (\lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil + \lambda_J * |\Delta_J|) \\ &\geq b_J * ((M_J - \lambda_J * b_J * |\Delta_J|)/b_J + \lambda_J * |\Delta_J|) \;=\; M_J \;.\end{aligned}$$

On the other hand,

$$\begin{aligned}|SP_J| &= b_J * (\lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil + \lambda_J * |\Delta_J|) \qquad (15) \\ &\leq b_J * ((M_J - \lambda_J * b_J * |\Delta_J|)/b_J + 1 + \lambda_J * |\Delta_J|] \;=\; M_J + b_J.\end{aligned}$$

■

Theorem 5 indicates a desirable property of the above sampling procedure, namely, at most b_J extra sample queries may exist in the sample. In fact, for our case $b_J \equiv 1$, it is easy to see $|SP_J| = M_J$ by substituting 1 for b_J in (15), i.e., no extra sample queries.

The above sampling procedure also adopts a mixture of simple random sampling, stratified sampling and cluster sampling. The frame H_J is first stratified into two strata: one H'_J composed of representative queries associated with joining column pairs in $\widetilde{\Delta}_J^{(2)}$ and the other H''_J composed of representative queries associated with joining column pairs in $\Delta_J - \widetilde{\Delta}_J^{(2)}$. Each stratum is further stratified into smaller strata, each of which consists of representative queries associated with one representative key conjunct type for an eligible joining column pair. Each second level stratum is then decomposed into several levels of smaller and smaller clusters according to the auxiliary column pair, the comparison operator in the left auxiliary predicate, the constant in the left auxiliary predicate, the comparison operator in the right auxiliary predicate, the constant in the right auxiliary predicate, and the project list of a representative query. For each second level stratum (with an eligible joining column pair $(R_i.a_n, R_j.a_m)$) for H'_J, we draw a sample of first level clusters with size $\lambda_J + 1$ by randomly choosing $\lambda_J + 1$ auxiliary column pairs[5] from $\Delta_{JA}|_{(R_i,R_j)}$. For each chosen first level cluster, a simple random sample of size 1 is drawn from each higher level (smaller) clusters. In other words, one sample query is to be obtained for each chosen first level cluster. The sample drawn from the second level stratum consists of these $\lambda_J + 1$ sample queries. The sample of queries drawn from H'_J is the union of all samples drawn from its second level strata. In a similar manner, we draw λ_J sample queries from each stratum for H''_J, and the sample drawn from H''_J is the union of the samples for these strata. The final sample is the union of the sample from H'_J and the sample from H''_J.

In fact, Case 1 can be unified with Case 2. If $|\Delta_J| * b_J > M_J$, then $\lambda_J = \lfloor M_J/(|\Delta_J| * b_J) \rfloor = 0$, and $|\widetilde{\Delta}_J^{(2)}| = \lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil = \lceil M_J/b_J \rceil$. The sampling procedure in Case 2 using such λ_J and $\widetilde{\Delta}_J^{(2)}$ agrees with the sampling procedure in Case 1 using $\widetilde{\Delta}_J^{(1)} = \widetilde{\Delta}_J^{(2)}$. Since $\lambda_J * b_J = 0$, no sample queries are drawn by the sampling procedure in Case 2 for any eligible joining column pair in $\Delta_J - \widetilde{\Delta}_J^{(2)}$, and one $(\lambda_J + 1 = 1)$ sample query is drawn for each representative key conjunct type associated with an eligible joining column pair in $\widetilde{\Delta}_J^{(2)}$. If $|\Delta_J| * b_J = M_J$, then $\lambda_J = \lfloor M_J/|\Delta_J| * b_J \rfloor = 1$, and $|\widetilde{\Delta}_J^{(2)}| = \lceil (M_J - \lambda_J * b_J * |\Delta_J|)/b_J \rceil = 0$. The sampling procedure in Case 2 using such λ_J and $\widetilde{\Delta}_J^{(2)}$ agrees with the sampling procedure in Case 1 using $\widetilde{\Delta}_J^{(1)} = \Delta_J$. Since $\widetilde{\Delta}_J^{(2)} = \emptyset$, all sample queries are drawn for eligible pairs in $\Delta_J - \widetilde{\Delta}_J^{(2)} = \Delta_J = \widetilde{\Delta}_J^{(1)}$. It is easy to see that Theorems 4 and 5 still hold for such extended λ_J and $\Delta_J^{(2)}$.

5. Derivation of Cost Estimation Formulas

After queries are classified and a sample is drawn from each query class, a cost estimation formula needs to be derived for each query class based on observed costs of sample queries. Such a cost formula includes a set of variables that affect costs of queries and a number of coefficients that reflect performance behavior of the underlying DBMS. How to derive a good cost estimation formula for a query class is the topic to be discussed in this section.

There are several variables that affect the cost of a query. Multiple regression in statistics allows us to establish a statistical relationship between the costs of queries and the relevant *affecting (explanatory) variables*. Such a statistical relationship can be used as a cost estimation formula for queries in a query class. We explore this idea in more details in the following subsections.

5.1. Identification of Explanatory Variables

It is not difficult to see that the following types of factors usually affect the cost of a query:

1. *The cardinality of an operand table*. The larger the cardinality of an operand table is, the higher the query (execution) cost. This is because the number of I/O's required to scan the operand table or its index(es) usually increases with the cardinality of the table.

2. *The cardinality of the result table*. A large result table implies that many tuples need to be processed, buffered, stored and transferred during query processing. Hence, the larger the result table is, the higher the corresponding query cost. Note that the cardinality of the result table is determined by the selectivity of (the qualification of) the query. This factor can be considered as the same as the selectivity of a query.

3. *The size of an intermediate result.* For a join query, if its qualification contains one or more conjunctive terms that refer to only one of its operand tables, called *separable conjunctive terms*, they can be used to reduce the relevant operand table before further processing is performed. The smaller the size of such an intermediate table is, the more efficient the query processing would be. For a unary query, if it can be executed by an index scan method, the query processing can be viewed as having two stages: the first stage is to retrieve the tuples via an index(es), the second stage is to check the retrieved tuples against the remaining conditions in the qualification. The number of tuples that are retrieved in the first stage can be considered as the size of the intermediate result for such a unary query.

4. *The tuple length of an operand table*[6]. This factor affects data buffering and transferring cost during query processing. However, this factor is usually not as significant as the above types of factors. It becomes important when the tuple lengths of tables in a database vary widely; for example, when multimedia data is stored in the tables.

5. *The tuple length of the result table*. Similar to the above factor, this factor affects data buffering and transferring cost, but it is not as significant as the first three types of factors. It may become important when it varies significantly from one query to another. Frequently, if the cardinalities of result tables for the queries in a query class do not change much, the tuple lengths of result tables may become more significant.

6. *The physical sizes (i.e., the numbers of occupied disk/buffer blocks) of operand tables and result tables*. Although the factors of this type are obviously controlled by the factors of types 1, 2, 4 and 5, they may reflect additional information, such as the percentage of free space assigned to an operand table (or a result table) and a combined effect of the previous factors.

7. *Contention in the system environment.* The factors of this type include contention for CPU, I/O, data items, and servers, etc. Obviously, such factors affect the performance

of a query. However, they are difficult to measure. The number of concurrent processes, the memory resident set sizes (RSS) of processes, and some other information about processes that we could obtain can only reflect part of all contention factors. That is why contention factors are usually omitted from existing cost models.

8. *The characteristics of an index*, such as index clustering ratio, the height and number of leaves of an index tree, the number of distinct values of an indexed column, and so on. If all tuples with the same index key value are physically stored together, the index is called as a clustered index, which has the highest index clustering ratio. For a referenced index, how the tuples with the same index key value are scattered in the physical storage has an obvious effect on the performance of a query. Other properties of an index, such as the height of the tree and the number of distinct key values, also affect the performance of a query.

The variables representing the above factors are the possible explanatory variables to be included in a cost formula.

5.2. *Development of Regression Cost Models*

As mentioned before, a multiple regression model is to be developed for estimating costs of queries in a query class. To develop such a regression model, we need to decide what explanatory variables to be included and how to include them in the model.

5.2.1. Variables Inclusion Principle In general, not all explanatory variables in Section 5.1 are included in a cost model. Some variables may not be significant for a particular model, while some others may not be available at the global level in an MDBS. Our general principle for including variables in a cost model is to include important variables and omit insignificant or unavailable variables.

Among the factors discussed in Section 5.1, the first three types of factors are often more important. The variables representing them are usually included in a cost model. The factors of types 4 and 5 are less important. The representing variables are included in a cost model only if they are significant. The variables representing factors of type 6 are included in a cost model if they are not dominated by other included variables. The variables representing the last two types of factors will be omitted from our cost models because they are usually not available at the global level in an MDBS. In fact, we assume that contention factors in a considered environment are approximately stable. Under this assumption, the contention factors are not very important in a cost model. The variables representing the characteristics of referenced indexes can possibly be included in a cost model if they are available and significant.

How to apply this variables inclusion principle to develop a cost model for a query class will be discussed in more details in the following subsection. Let us first give some notations for the variables.

Let R_U be the operand table for a unary query; R_{J1} and R_{J2} be the two operand tables for a join query; N_U, N_{J1} and N_{J2} be the cardinalities of R_U, R_{J1} and R_{J2}, respectively; L_U, L_{J1} and L_{J2} be the tuple lengths of R_U, R_{J1} and R_{J2}, respectively; RL_U and RL_J be the tuple lengths of the result tables for the unary query and the join query, respectively. Let S_U and S_J be the selectivities of the unary query and the join query, respectively; S_{J1} and S_{J2} be the selectivities of the conjunctions of all separable conjunctive terms for R_{J1} and R_{J2}, respectively; S_{U1} be the selectivity[7] of an index-usable key predicate (conjunct), if applicable, of the unary query.

5.2.2. Regression Models for Unary Query Classes Based on the inclusion principle, we divide a regression model to be developed for a unary query class into two parts:

$$regression\ model \;=\; basic\ model \;+\; secondary\ part\ . \tag{16}$$

The basic model is the essential part of the regression model, while the secondary part is used to further improve the model.

The set V_{UB} of potential explanatory variables to be included in the basic model contains the variables representing the factors of types $1 \sim 3$. By definition, $TN_U = N_U * S_{U1}$ and $RN_U = N_U * S_U$ are the cardinalities of the intermediate table and result table for the unary query, respectively. Therefore, $V_{UB} = \{\, N_U,\ TN_U,\ RN_U \,\}$.

If all potential explanatory variables in V_{UB} are chosen, the basic model is

$$Y \;=\; B_0 \;+\; B_1 * N_U \;+\; B_2 * TN_U \;+\; B_3 * RN_U. \tag{17}$$

As to be discussed later, some potential variable(s) may be insignificant for a given query class and, therefore, is not included in the basic model. For the common query class $G_{1\ 5}$, $B_2 \equiv 0$ since there is no index-usable key predicate (conjunct) for a query in the class.

The basic model captures the major performance behavior of queries in a query class. It will also be used to detect and delete noises of raw sample data (see Section 5.3). In fact, the basic model (17) is based on existing cost models [7, 9, 18, 21, 22, 23] for a DBMS. The parameters $B_0,\ B_1,\ B_2$ and B_3 in (17) could be interpreted as the initialization cost, the cost of retrieving a tuple from the operand table, the cost of using the index referenced by the key predicate to fetch a tuple, and the cost of processing a result tuple, respectively. In a traditional cost model, a parameter may be split up into several parts (e.g., B_1 may consist of I/O cost and CPU cost) and can be determined by analyzing the implementation details of the employed access method. However, in an MDBS, the implementation details of access methods are usually not known to the global query optimizer. The parameters are, therefore, estimated by multiple regression based on sample queries instead of an analytical method.

To further improve the basic model, some secondary explanatory variables may be included into the model. The set V_{US} of potential explanatory variables for the secondary part of a model contains the variables representing factors of types $4 \sim 6$. The real physical sizes of the operand table and result table of a unary query may not be known exactly in an MDBS. However, they[8] can be estimated by $Z_U = N_U * L_U$ and $RZ_U = RN_U * RL_U$, respectively. To simplify description, we call Z_U and RZ_U the operand table length and result table length, respectively. Therefore, $V_{US} = \{\, L_U,\ RL_U,\ Z_U,\ RZ_U \,\}$. Any other variables, if available, could also be included in V_{US}.

If all potential variables in V_{US} are added to (17), the full regression model is

$$\begin{aligned} Y \;=\; & B_0 \;+\; B_1 * N_U \;+\; B_2 * TN_U \;+\; B_3 * RN_U \;+\; B_4 * L_U \\ & + B_5 * RL_U \;+\; B_6 * Z_U \;+\; B_7 * RZ_U\ . \end{aligned} \tag{18}$$

For a specific query class, not all variables, especially the secondary explanatory variables, are necessary for its regression model. How to select significant explanatory variables in the regression model for a query class will be discussed in Section 5.2.4.

Note that, for some query class, a variable might appear in its regression model in another form. For example, if the access method for a query class sorts the operand table of a query based on a column(s) before performing further processing, some terms like $N_U * log\ N_U$ and/or $log\ N_U$ could be included in its regression model. Let a new variable represent such a term. This new variable may replace an existing variable in $V_{UB} \cup V_{US}$ or be an additional secondary variable in V_{US}. A regression model can be adjusted according to any available information about the relevant access method.

5.2.3. Regression Models for Join Query Classes Similar to a unary query class, the regression model for a join query class consists of a basic model plus a possible secondary part.

The set V_{JB} of potential explanatory variables for the basic model contains the variables representing factors of types $1 \sim 3$. By definition, $RN_J = N_{J1} * N_{J2} * S_J$ is the cardinality of the result table for a join query; $TN_{J1} = N_{J1} * S_{J1}$ is the size of the intermediate table obtained by performing the conjunction of all separable conjunctive terms on R_{J1}; $TN_{J2} = N_{J2} * S_{J2}$ is the size of the intermediate table obtained by performing the conjunction of all separable conjunctive terms on R_{J2}; $TN_{J12} = TN_{J1} * TN_{J2}$ is the size of the Cartesian product of the intermediate tables. Therefore, $V_{JB} = \{ N_{J1},\ N_{J2},\ TN_{J1},\ TN_{J2},\ TN_{J12},\ RN_J \}$.

If all potential explanatory variables in V_{JB} are selected, the basic model is

$$Y = B_0 + B_1 * N_{J1} + B_2 * N_{J2} + B_3 * TN_{J1} + B_4 * TN_{J2} + B_5 * TN_{J12} + B_6 * RN_J. \tag{19}$$

Similar to a unary query class, this basic model is based on existing cost models for a DBMS. The parameters $B_0,\ B_1,\ B_2,\ B_3,\ B_4,\ B_5$ and B_6 could be interpreted as the initialization cost, the cost of pre-processing a tuple in the first operand table, the cost of pre-processing a tuple in the second operand table, the cost of retrieving a tuple from the first intermediate table, the cost of retrieving a tuple from the second intermediate table, the cost of processing a tuple in the Cartesian product of the two intermediate tables, and the cost of processing a result tuple, respectively.

The basic model may be further improved by including some additional beneficial variables. The set V_{JS} of potential explanatory variables for the secondary part of a model contains the variables representing factors of types $4 \sim 6$. Similar to unary queries, the physical size of a table is represented by the table length. In other words, the physical sizes of the first operand table, the second operand table and the result table are represented by the variables: $Z_{J1} = N_{J1} * L_{J1}$, $Z_{J2} = N_{J2} * L_{J2}$, $RZ_J = RN_J * RL_J$, respectively. Therefore, $V_{JS} = \{ L_{J1},\ L_{J2},\ RL_J,\ Z_{J1},\ Z_{J2},\ RZ_J \}$. Any other useful variables, if available, could also be included in V_{JS}.

If all potential explanatory variables in V_{JS} are added to (19), the full regression model is

$$Y = B_0 + B_1 * N_{J1} + B_2 * N_{J2} + B_3 * TN_{J1} + B_4 * TN_{J2}$$

$$+ B_5 * TN_{J12} + B_6 * RN_J + B_7 * L_{J1} + B_8 * L_{J2}$$
$$+ B_9 * RL_J + B_{10} * Z_{J1} + B_{11} * Z_{J2} + B_{12} * RZ_J . \quad (20)$$

Note that further improvements on cost estimates produced by a cost model could be achieved by properly choosing which operand table in a join query to be the first operand table and which one to be the second operand table. For example, a good criterion for choosing the table order for the query class G_{21} could be choosing the operand table that can be accessed via a clustered-indexed joining column as the first table. When both operand tables of a query can be accessed via the same physical access path, e.g., sequential scan, or index scan, one could simply choose the left join table as the first operand table or choose the smaller table as the first operand table. Note that different component database systems may employ different criteria. One can try different criteria and choose the one that yields better estimates.

Similar to a unary query class, not all variables in V_{JB} and V_{JS} are necessary for a join query class. A procedure to choose significant variables in a model will be described in the following subsection. In addition, some additional variables may be included, and some variables could be included in another form. In general, a regression model can be adjusted according to the real situation. The more information is available, the better a regression model could be derived.

5.2.4. Selection of Variables for Regression Models To determine the variables to be included in a regression model, one approach is to evaluate all possible subset models and choose the best one(s) among them according to some criterion. However, evaluating all possible models may not be practically feasible when the number of variables is large.

To reduce the amount of computation, two types of selection procedures have been proposed [3]: the forward selection procedure and the backward elimination procedure. The forward selection procedure starts with a model containing no variables, i.e., only a constant term, and introduces explanatory variables into the regression model one at a time. The backward elimination procedure starts with the full model and successively drops one explanatory variable at a time. Both procedures need a criterion for selecting the next explanatory variable to be included in or removed from the model and a condition for stopping the procedure. With k variables, these procedures will involve evaluation of at most $(k + 1)$ models as contrasted with the evaluation of 2^k models necessary for examining all possible models.

To select a suitable regression model for a query class, we use a mixed forward and backward procedure described below (see Figure 4). We start with the basic model including all variables in the set (V_{UB} or V_{JB}) of basic explanatory variables for the query class. We apply the backward elimination procedure first to drop some insignificant terms (explanatory variables) from the model. We then apply the forward selection procedure to find additional significant explanatory variables from the set (V_{US} or V_{JS}) of secondary explanatory variables for the query class.

The next explanatory variable X to be removed from the basic model during the first backward stage is the one that (1) has the smallest simple correlation coefficient[9] with the response variable Y and (2) makes the reduced model (i.e., the model after X is removed) have a smaller standard error of estimation[10] than the original model or the two standard

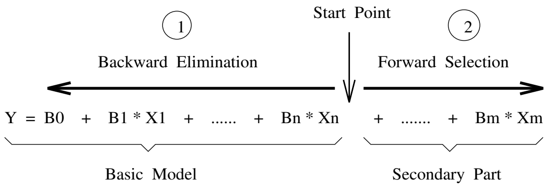

Figure 4. Selection of Variables for Regression Model

errors of estimation very close to each other, for instance, within 1% relative error. If the next explanatory variable satisfying (1) does not satisfy (2), or no more explanatory variable is in the model, the backward elimination procedure ends. Condition (1) chooses the variable which usually contributes the least among other variables in predicting Y. Condition (2) guarantees that removing the chosen variable results in an improved model or affects the model only very little. Removing the variables that affect the model very little can reduce the complexity and maintenance overhead of the model.

The next explanatory variable X to be added into the current model during the second forward stage is the one that (a) is in the set of secondary explanatory variables; (b) has the largest simple correlation coefficient with the response variable Y that has been adjusted for the effect of the current model (i.e., the largest simple correlation coefficient with the residuals of the current model); and (c) makes the augmented model (i.e., the model that includes X) have a smaller standard error of estimation than the current model and the two standard errors of estimation not very close to each other, for instance, greater than 1% relative error. If the next explanatory variable satisfying (a) and (b) does not satisfy (c), or no more explanatory variable exists, the forward selection procedure ends. The reasons for using conditions (a) $\sim$ (c) are similar to the situation for removing a variable. In particular, a variable is not added into the model unless it improves the standard error of estimation significantly in order to reduce the complexity of the model.

Since we start with the basic model, which has a high possibility to be the appropriate model for the given query class, the backward elimination and forward selection will most likely stop soon after they are initiated. Therefore, our procedure is likely more efficient than a pure forward or backward procedure. However, in the worst case, the above procedure will still check $(k + 1)$ models for k potential explanatory variables, which is the same as a pure forward or backward procedure.

Once the explanatory variables in a regression model have been selected, the regression coefficients B's can be estimated by the *method of least squares (LS)*. The resulting *fitted regression equation* can be used as the cost estimation formula for the relevant query class.

5.3. Measures for Developing Useful Models

To develop a useful regression model, measures need to be taken during query sampling and model development. Furthermore, a developed regression model should be validated before it is used. Improvements may be needed if the model proves unacceptable.

Sample Size. To develop a good regression model, we need to use a sample with a sufficient size. What sample size is sufficient depends on the number of explanatory variables involved in the model. For a regression model with k explanatory variables, there are $(k + 2)$ parameters that need to be estimated: the $k + 1$ regression coefficients and the variance of error terms. A commonly used rule for sampling is to sample at least 10 observations for every parameter to be estimated [17]. However, a regression model is not pre-determined in our application. It is expected that most variables in V_B will be selected and only a few variables in V_S will be used. Therefore, we expect that no more than $|V_B| + \lceil |V_S|/2 \rceil$ variables will be included in a cost model in most cases. For simplicity, we draw a sample of queries with a minimum size $M = 10 * (|V_B| + \lceil |V_S|/2 \rceil + 2)$ from a query class. As we have proved, the sampling procedures discussed in Section 4 can guarantee a sample to have the required minimum sample size M.

Outliers. Outliers are extreme observations. In a residual plot, outliers are the points that lie far beyond the scatter of the majority of points. Frequently, an outlier results from a mistake or other extraneous causes. A fitted equation may be pulled disproportionately towards an outlying observation under the method of least squares. Since our objective is to derive a cost estimation formula that is good for the majority of queries in a query class, we use the set of observations with outliers removed for derivation of the cost formula.

Multicollinearity. When the explanatory variables are highly correlated among themselves, *multicollinearity* among them is said to exist. The presence of multicollinearity does not, in general, inhibit our ability to obtain a good fit nor does it tend to affect predictions of new observations, provided these predictions are made within the region of observations. However, the estimated regression coefficients tend to have large sampling variability. To make reasonable predictions beyond the region of observations and obtain more precise information about the true regression coefficients, it is better to avoid multicollinearity among explanatory variables.

A method to detect the presence of multicollinearity that is widely used is by means of *variance inflation factors (VIF)* [15]. These factors measure how much the variances of the estimated regression coefficients are inflated as compared to when the explanatory variables are not linearly related. Explanatory variables with large variance inflation factors are removed from a model. This method is also used in our system.

Validation of Model Assumptions. A regression model has the following three assumptions [15]: (1) the error terms are uncorrelated; (2) the error terms have the same variance; and (3) the error terms are normally distributed. In general, regression analysis is not seriously affected by slight to moderate departures from the assumptions. The assumptions can be ranked in terms of the seriousness of the failure of the assumption to hold from the most serious to the least serious as follows: assumptions (1), (2) and (3).

For our application, the observed costs of repeated executions of a sample query have no inherent relationship with the observed costs of repeated executions of another sample query under the assumption that the contention factors in the system are approximately stable. Hence the first assumption should be satisfied. This is a good property because the violation of assumption (1) is the most serious to a regression model.

However, the variance of the observed costs of repeated executions of a sample query may increase with the level (magnitude) of query cost. This is because the execution of a sample query with longer time (larger cost) may suffer more disturbances in the system than the execution of a sample query with shorter time. Thus assumption (2) may be violated in our regression models. A statistical hypothesis-test based on the Spearman's rank correlation coefficient [17] is used to detect such a violation. If such a violation happens, an iterative weighted least squares (WLS) procedure [15] is adopted to remedy the problem.

The observed costs of repeated executions of a sample query may not follow the normal distribution; i.e., assumption (3) may not hold. The observed costs are usually skewed to the right because the observed costs stay at a stable level for most time and become larger from time to time when disturbances occur in the system. Fortunately, many studies have shown that regression analysis is robust to the normality assumption [15, 17]; that is, the technique will give usable results even if this assumption is not satisfied.

Testing Significance of Regression Model. To evaluate the goodness of the developed regression model, two descriptive measures are used: the standard error of estimation s and the coefficient of multiple determination[11] R^2. A good regression model is evidenced by a small standard error of estimation and a high coefficient of multiple determination.

The significance of the developed model is further tested by using the F-test [15, 17] in our system. In addition, some test queries are performed on the underlying DBMS. Their observed costs and the estimated costs given by using the developed model are compared. A model is accepted only it can give acceptable cost estimates for the majority of test queries.

5.4. *Complete Statistical Procedure*

Figure 5 shows a complete statistical procedure for deriving cost estimation formulas for query classes on a component DBMS. Although the procedure can be logically viewed to have three tasks: query classification, query sampling and regression analysis, the tasks may need to be iterated many times before satisfactory cost estimation formulas are obtained.

6. Experimental Results

To check the practical feasibility of the method, experiments were conducted. Experimental results are reported in this section. As we will see, the experimental results demonstrate that the query sampling method is quite promising in estimating local cost parameters in an MDBS environment.

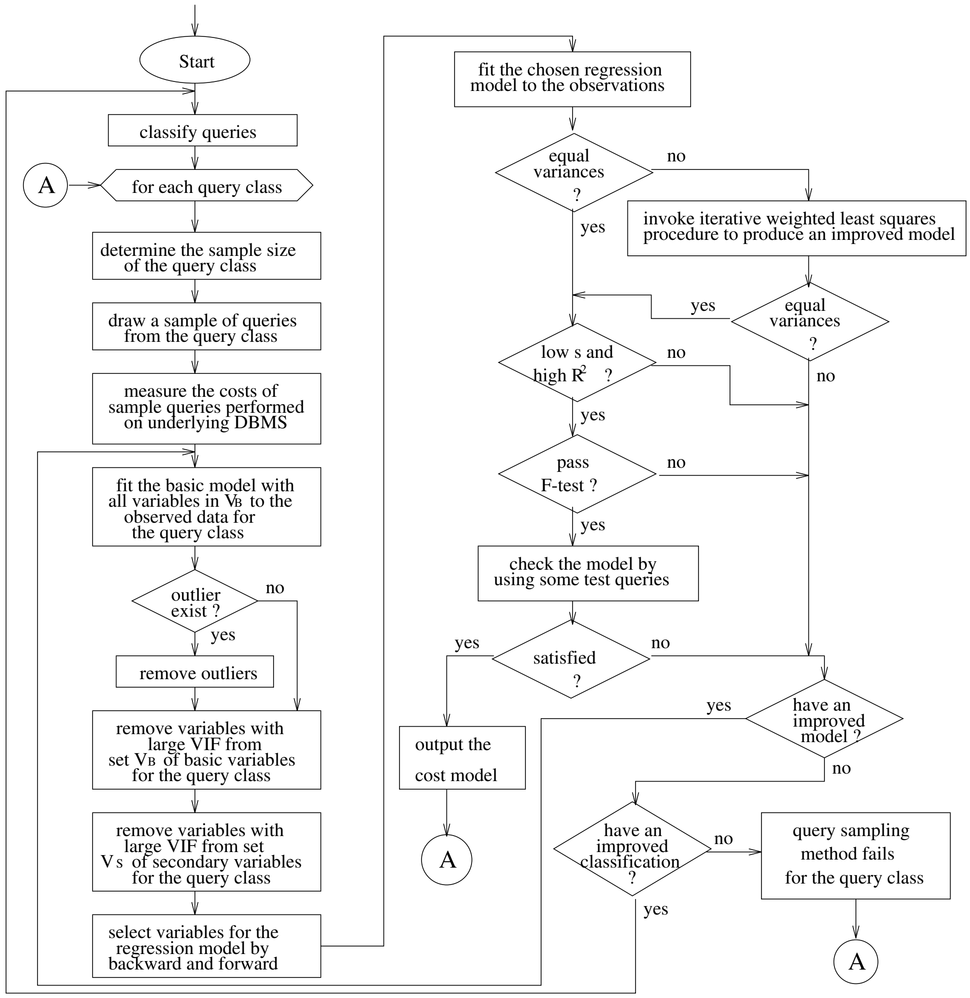

Figure 5. The Complete Statistical Procedure for Developing Cost Models

6.1. Experimental Environment

Our MDBS prototype CORDS-MDBS was used as a multidatabase environment for the experiments. Each component database system in CORDS-MDBS contains a database, a DBMS, and an MDBS agent that provides the global MDBS server with a uniform relational ODBC (Open Database Connectivity) interface. All component queries coming from the global MDBS server are passed to the relevant MDBS agents in the component DBSs

Three commercial DBMSs, i.e., ORACLE 7.0, EMPRESS 4.6 and DB2/6000 1.1.0, were used as component DBMSs in the experiments. All the component DBMSs were run on IBM RS/6000 model 220 machines (see Figure 6).

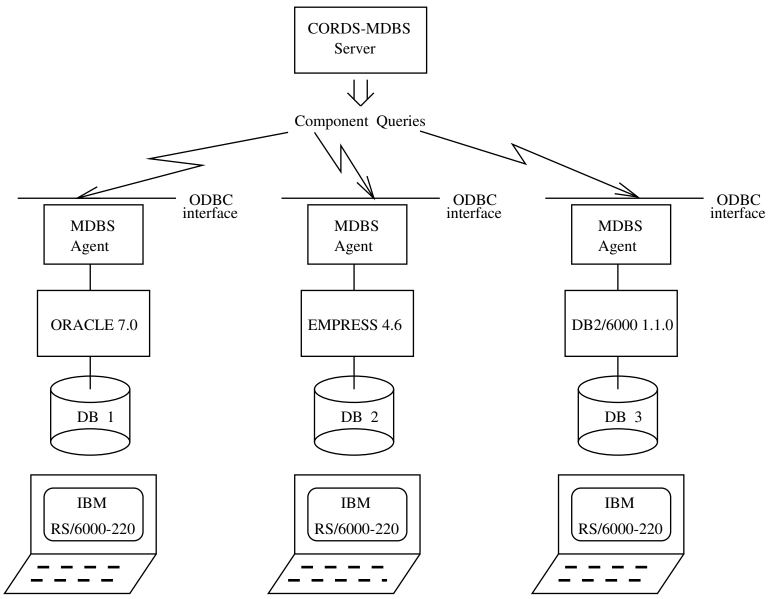

Figure 6. Experimental Environment

The experiments were conducted in a system environment where the contention factors were approximately stable. For example, they were performed during midnights and weekends when there was no or little interference from other users in the systems. However, occasional interference from other users still existed since the systems are shared resources.

6.2. Experimental Databases

A simple component database with diverse characteristics was created for each component DBS in the experiments. Let DB_1, DB_2 and DB_3 denote the component databases managed by ORACLE, EMPRESS and DB2/6000, respectively. Each component database consists of 12 tables with all integer columns, as shown in Table 5. Some of the columns in a table are indexed or clustered-indexed. Since a user cannot declare a clustered-indexed column for a table managed by EMPRESS 4.6 or DB2/6000 1.1.0, the clustered-indexed columns in Table 5 were specified as regular indexed columns in the relevant component databases. The cardinalities of the tables in DB_1, DB_2 and DB_3 are shown in Table 6. Data in the tables of the component databases are randomly generated from various ranges.

Table 5. Tables in Experimental Component Databases

No.	Table	Indexed Col.	Clustered-Indexed Col.
1	$R_1(a_1, a_2, a_3)$	a_2, a_3	
2	$R_2(a_1, a_2, a_3, a_4, a_5)$	a_2, a_4	a_3
3	$R_3(a_1, a_2, a_3, a_4, a_5)$	a_3, a_4	
4	$R_4(a_1, a_2, a_3, a_4, a_5, a_6, a_7)$	a_2, a_6	
5	$R_5(a_1, a_2, a_3, a_4, a_5, a_6, a_7)$	a_2, a_4	a_1
6	$R_6(a_1, a_2, a_3, a_4, a_5, a_6, a_7)$	a_4, a_7	a_1
7	$R_7(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9)$	a_1, a_4, a_7	a_2
8	$R_8(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9)$	a_2, a_6, a_9	a_3
9	$R_9(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9)$	a_1, a_3, a_5, a_8	
10	$R_{10}(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11})$	a_2, a_4, a_{10}	a_1
11	$R_{11}(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11})$	a_2, a_7	a_1
12	$R_{12}(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9,$ $a_{10}, a_{11}, a_{12}, a_{13})$	a_2, a_5, a_{11}, a_{13}	

Table 6. Table Cardinalities in Experimental Component Databases

Database	Table Cardinalities					
	$\lvert R_1\rvert$	$\lvert R_2\rvert$	$\lvert R_3\rvert$	$\lvert R_4\rvert$	$\lvert R_5\rvert$	$\lvert R_6\rvert$
DB_1	25000	20000	1700	300	3000	1000
DB_2	2500	2000	170	30	300	100
DB_3	25000	20000	1700	300	3000	1000
	$\lvert R_7\rvert$	$\lvert R_8\rvert$	$\lvert R_9\rvert$	$\lvert R_{10}\rvert$	$\lvert R_{11}\rvert$	$\lvert R_{12}\rvert$
DB_1	5000	15000	4000	10000	700	8000
DB_2	500	1500	400	1000	70	800
DB_3	5000	15000	4000	10000	700	8000

One may ask why a real-world database was not adopted for our experiments. There are several reasons for choosing the above synthetic experimental databases instead of a real-world database:

- First, using random data is a typical approach for simulation experiments because experimental data can be designed to demonstrate various characteristics. The characteristics of a working commercial database cannot be controlled. They are restricted to a particular application. The above experimental databases, however, were designed to have diverse characteristics such as various table degrees, table cardinalities, indexed columns, index clustering degrees, and selectivities for different columns. We felt that it would be more valuable to test if our method can handle such a database with diverse characteristics. It is expected that the method would behave better if it is applied to a database with more homogeneous characteristics.

- Second, it is always questionable why a particular real-world database is chosen instead of others. However, it is impossible to test them all. A synthetic database incorporating characteristics from several databases appears an appropriate solution to this problem.
- Third, it is difficult to obtain a real-world database instance with a reasonable size in a university environment. It is not very useful to estimate query costs for a database with a few tuples in each table because there is no much difference among the costs. However, making up hundreds and thousands of meaningful tuples for a real-world table is time-consuming and not valuable. Using randomly-generated data greatly simplifies creating databases with known characteristics.

Therefore, the above experimental databases were adopted in our experiments.

6.3. Test Queries

Test queries used to check the goodness of a cost estimation formula derived for a query class are randomly generated from the query class in a similar way as sample queries. However, the forms of test queries are more general than those of sample queries in the query class.

The test queries for a unary query class ($G_{1\ 1}$, $G_{1\ 2}$, $\cdots$ or $G_{1\ 5}$) are of the following forms:

$$\pi_{\alpha(i)}(\sigma_{R_i.a_n\ \omega\ C^{(i.a_n)}}(R_i))\ ,$$

$$\pi_{\alpha(i)}(\sigma_{R_i.a_n\ \omega\ C^{(i.a_n)}\ \wedge\ R_i.a_m\ \omega_1\ C^{(i.a_m)}}(R_i))\ ,$$

$$\pi_{\alpha(i)}(\sigma_{R_i.a_n\ \omega\ C^{(i.a_n)}\ \wedge\ (\ R_i.a_p\ \omega_2\ C^{(i.a_p)}\ \vee\ R_i.a_q\ \omega_3\ C^{(i.a_p)})}(R_i))\ ,$$

where $\omega_1,\ \omega_2,\ \omega_3\ \in\ \{\ =,\ \neq,\ <,\ \leq,\ >,\ \geq\ \}$; $\omega \in \{\ =\ \}$ for $G_{1\ 1}$ and $G_{1\ 2}$; $\omega \in \{\ <,\ \leq,\ >,\ \geq\ \}$ for $G_{1\ 3}$ and $G_{1\ 4}$; $\omega \in \{\ =,\ \neq,\ <,\ \leq,\ >,\ \geq\ \}$ for $G_{1\ 5}$. The qualifications in the test queries satisfy the requirements of the corresponding query class; for example, $R_i.a_n$ is clustered-indexed for $G_{1\ 1}$.

The test queries for a join query class ($G_{2\ 1}$, $G_{2\ 2}$, $G_{2\ 3}$, or $G_{2\ 4}$) are of the following forms:

$$\pi_{\alpha(i\ j)}(R_i \underset{qualification}{\bowtie} R_j)$$

where:

$$\begin{aligned}
qualification\ &::=\ [\ (simple_left \mid disjunct_left)\ \wedge\ ]\ key_join \\
&\quad\ [\ \wedge\ \ (simple_right \mid disjunct_right)\ ]\ [\ \wedge\ additional_join\ ] \\
key_join\ &::=\ R_i.a_n = R_j.a_m \\
simple_left\ &::=\ R_i.a_p\ \omega_1\ C^{(i.a_p)} \\
simple_right\ &::=\ R_j.a_q\ \omega_2\ C^{(j.a_q)} \\
disjunct_left\ &::=\ R_i.a_u\ \omega_3\ C^{(i.a_u)}\ \vee\ R_i.a_v\ \omega_4\ C^{(i.a_v)} \\
disjunct_right\ &::=\ R_j.a_s\ \omega_5\ C^{(j.a_s)}\ \vee\ R_j.a_t\ \omega_6\ C^{(j.a_t)} \\
additional_join\ &::=\ R_i.a_x = R_j.a_y
\end{aligned}$$

here $\omega_1 \sim \omega_6 \in \{ =, \neq, <, \leq, >, \geq \}$, and each qualification satisfies the requirements of the given query class; for example, at least one of $R_i.a_n$ and $R_j.a_m$ is clustered-indexed for $G_{2\ 1}$.

6.4. *Experimental Results*

In the experiments, sample queries are drawn from each query class, as described in Section 4. Sample queries are performed on the three component database systems. Their observed costs are used to derive cost estimation formulas for the relevant query classes by multiple regression as described in Section 5.

Tables 7 $\sim$ 12 show the derived cost formulas and the relevant statistical measures. It can be seen that:

Table 7. Derived Cost Formulas for Query Classes on ORACLE 7.0

query class	Cost Estimation Formula
$G_{1\ 1}$	0.675963e-1 $+$ 0.388098e-2 $* RN_U$ $+$ 0.831587e-2 $* RL_U$
$G_{1\ 2}$	0.866475e-1 $+$ 0.177483e-2 $* TN_U$ $+$ 0.926299e-2 $* RN_U$ $+$ 0.443237e-6 $* Z_U$
$G_{1\ 3}$	0.146923 $+$ 0.335288e-3 $* TN_U$ $+$ 0.350591e-2 $* RN_U$
$G_{1\ 4}$	0.354301 $+$ 0.105255e-2 $* TN_U$ $+$ 0.32336e-2 $* RN_U$ $+$ 0.852187e-4 $* RZ_U$
$G_{1\ 5}$	0.16555 $+$ 0.149208e-3 $* N_U$ $+$ 0.307219e-2 $* RN_U$ $+$ 0.105712e-3 $* RZ_U$
$G_{2\ 1}$	0.149939 $+$ 0.153634e-2 $* TN_{J2}$ $+$ 0.400375e-7 $* TN_{J12}$ $+$ 0.401116e-2 $* RN_J$
$G_{2\ 2}$	0.192209 $+$ 0.161011e-2 $* TN_{J2}$ $+$ 0.573257e-7 $* TN_{J12}$ $+$ 0.426256e-2 $* RN_J$
$G_{2\ 3}$	0.176158 $+$ 0.951479e-3 $* TN_{J12}$
$G_{2\ 4}$	-0.236703e-1 $+$ 0.143572e-3 $* N_{J2}$ $+$ 0.61871e-3 $* TN_{J1}$ $+$ 0.680628e-3 $* TN_{J2}$ $+$ 0.399927e-6 $* TN_{J12}$ $+$ 0.316129e-2 $* RN_J$

Table 8. Statistical Measures for Cost Formulas on ORACLE 7.0

query class	coef. of multiple determ.	standard err. of estimation	average cost (sec.)	F-statistic (critical value at $\alpha = 0.01$)	Spearman's rank correlation (critical val. at $\alpha = 0.01$)	WLS ?
$G_{1\ 1}$	0.8361	0.438e-1	0.169e+0	0.204e+3 ($>$ 4.88)	0.185 ($<$ 0.257)	no
$G_{1\ 2}$	0.6568	0.106e+0	0.204e+0	0.568e+2 ($>$ 3.97)	0.054 ($<$ 0.243)	yes
$G_{1\ 3}$	0.9764	0.179e+1	0.816e+1	0.158e+4 ($>$ 4.89)	0.143 ($<$ 0.264)	yes
$G_{1\ 4}$	0.9675	0.274e+1	0.114e+2	0.116e+4 ($>$ 4.29)	0.210 ($<$ 0.213)	yes
$G_{1\ 5}$	0.9981	0.874e+0	0.136e+2	0.154e+5 ($>$ 3.97)	0.021 ($<$ 0.244)	yes
$G_{2\ 1}$	0.9920	0.343e+1	0.186e+2	0.489e+4 ($>$ 4.27)	0.126 ($<$ 0.212)	yes
$G_{2\ 2}$	0.9899	0.150e+1	0.607e+1	0.373e+4 ($>$ 4.28)	0.061 ($<$ 0.215)	yes
$G_{2\ 3}$	0.9246	0.516e+3	0.753e+3	0.148e+4 ($>$ 7.06)	0.074 ($<$ 0.211)	yes
$G_{2\ 4}$	0.9767	0.153e+1	0.713e+1	0.981e+3 ($>$ 3.52)	0.133 ($<$ 0.211)	yes

- Each cost formula involves a number of basic explanatory variables plus one possible secondary explanatory variable. Most of them contain a variable for the cardinality of the result table, i.e., RN_U, or RN_J; and most of them contain one or more variables for the sizes of intermediate results, such as TN_U, TN_{J2}, and TN_{J12}. However, many of them do not contain variable(s) for the cardinality(ies) of operand table(s). This observation indicates that the basic explanatory variables are more important than

Table 9. Derived Cost Formulas for Query Classes on EMPRESS 4.6

query class	Cost Estimation Formula
$G_{1\ 2}$	0.104265 $+$ 0.452447e-2 $* TN_U$ $+$ 0.463534e-2 $* RL_U$
$G_{1\ 4}$	0.129558 $-$ 0.487936e-3 $* N_U$ $+$ 0.292076e-2 $* TN_U$ + 0.524435e-3 $* RN_U$ + 0.381775e-4 $* Z_U$
$G_{1\ 5}$	0.121125 $+$ 0.272387e-2 $* N_U$ + 0.698155e-4 $* RN_U$ + 0.799805e-4 $* RZ_U$
$G_{2\ 2}$	0.391227 $-$ 0.533736e-3 $* N_{J1}$ + 0.127505e-1 $* TN_{J1}$ + 0.408462e-2 $* RN_J$ + 0.167088e-3 $* Z_{J1}$
$G_{2\ 3}$	0.367497 + 0.28452e-2 $* TN_{J12}$ + 0.881159e-3 $* RZ_J$
$G_{2\ 4}$	0.199148e+1 + 0.282653e-2 $* TN_{J12}$ + 0.112948e-3 $* Z_{J2}$ + 0.151314e-3 $* RZ_J$

Table 10. Statistical Measures for Cost Formulas on EMPRESS 4.6

query class	coef. of multiple determ.	standard err. of estimation	average cost (sec.)	F-statistic (critical value at $\alpha = 0.01$)	Spearman's rank correlation (critical val. at $\alpha = 0.01$)	WLS ?
$G_{1\ 2}$	0.7756	0.904e$-$2	0.137e+0	0.183e+3 ($>$ 4.78)	0.184 ($<$ 0.224)	no
$G_{1\ 4}$	0.9121	0.367e+0	0.110e+1	0.382e+3 ($>$ 2.16)	0.054 ($<$ 0.190)	yes
$G_{1\ 5}$	0.9991	0.579e$-$1	0.219e+1	0.347e+5 ($>$ 3.98)	0.043 ($<$ 0.246)	yes
$G_{2\ 2}$	0.9883	0.146e+1	0.621e+1	0.243e+4 ($>$ 3.83)	-0.010 ($>$ -0.214)	yes
$G_{2\ 3}$	0.9985	0.405e+2	0.330e+3	0.373e+5 ($>$ 4.72)	0.017 ($<$ 0.216)	yes
$G_{2\ 4}$	0.9997	0.329e+2	0.645e+3	0.111e+6 ($>$ 7.08)	0.027 ($<$ 0.217)	yes

Table 11. Derived Cost Formulas for Query Classes on DB2/6000

query class	Cost Estimation Formula
$G_{1\ 2}$	0.351467e-1 $+$ 0.165762e-6 $* N_U$ $+$ 0.1791e-2 $* TN_U$ $+$ 0.367672e-2$* RN_U$ $+$ 0.15752e-6$* Z_U$
$G_{1\ 4}$	0.234767e-1 $+$ 0.276511e-3 $* N_U$ + 0.25797e-2 $* RN_U$ + 0.243711e-3 $* RZ_U$
$G_{1\ 5}$	0.356131e-1 $+$ 0.283372e-3 $* N_U$ + 0.263305e-2 $* RN_U$ + 0.24169e-3 $* RZ_U$
$G_{2\ 2}$	0.306633 $+$ 0.148516e-2 $* TN_{J2}$ + 0.626496e-6 $* TN_{J12}$ + 0.405455e-2 $* RN_J$
$G_{2\ 3}$	-0.190759e-1 $+$ 0.417679e-4 $* N_{J1}$ $+$ 0.164259e-3 $* N_{J2}$ + 0.105605e-2 $* TN_{J1}$ + 0.107104e-2 $* TN_{J2}$ + 0.210705e-2 $* RN_J$ + 0.300418e-3 $* RZ_J$
$G_{2\ 4}$	0.446699 $+$ 0.314534e-3 $* N_{J2}$ + 0.555693e-3 $* TN_{J1}$ + 0.478572e-3 $* TN_{J2}$ + 0.789005e-6 $* TN_{J12}$ + 0.358929e-2 $* RN_J$

Table 12. Statistical Measures for Derived Cost Formulas on DB2/6000

query class	coef. of multiple determ.	standard err. of estimation	average cost (sec.)	F-statistic (critical value at $\alpha = 0.01$)	Spearman's rank correlation (critical val. at $\alpha = 0.01$)	WLS ?
$G_{1\ 2}$	0.8148	0.404e$-$1	0.104e+0	0.116e+3 ($>$ 3.64)	-0.005 ($>$ -0.222)	yes
$G_{1\ 4}$	0.9949	0.861e+0	0.904e+1	0.947e+4 ($>$ 2.66)	0.080 ($<$ 0.190)	yes
$G_{1\ 5}$	0.9995	0.441e+0	0.149e+2	0.639e+5 ($>$ 3.97)	0.218 ($<$ 0.244)	yes
$G_{2\ 2}$	0.9325	0.431e+1	0.931e+1	0.539e+3 ($>$ 4.29)	0.033 ($<$ 0.213)	yes
$G_{2\ 3}$	0.9458	0.975e+0	0.338e+1	0.334e+3 ($>$ 3.18)	-0.005 ($>$ -0.212)	yes
$G_{2\ 4}$	0.9594	0.244e+1	0.882e+1	0.552e+3 ($>$ 3.52)	0.132 ($<$ 0.211)	yes

the secondary ones, and the sizes of intermediate and final results are usually more significant among all the basic explanatory variables.

- As predicted before, a tuple length variable, such as RL_U, may be significant in the cost formula for a query class containing queries whose result tables are all small, for example, $G_{1\ 1}$ on DB_1, and $G_{1\ 2}$ on DB_2.

- The cost formulas for the same query class on different component database systems can be quite different, such as the cost formulas for $G_{2\ 3}$ on the three component DBSs.

- Most cost formulas capture over 90% variability in query cost, from observing the coefficients of total determination. The best one ($G_{2\ 4}$ on DB_2) captures about 99.97% variability, and the worst one ($G_{1\ 2}$ on DB_1) captures about 65.68% variability. The worse cases occur for $G_{1\ 1}$ and $G_{1\ 2}$ whose queries can be executed very fast, i.e., small-cost queries, due to their efficient access methods and small result tables.

- The standard errors of estimation for the cost formulas are acceptable, compared with the magnitudes of the relevant average observed costs of the sample queries. On the average, the standard error of estimation is about 23.8% of the corresponding average observed cost. The best standard error of estimation (for $G_{1\ 5}$ on DB_2) is only about 2.6% of the corresponding average observed cost. The worst standard error of estimation (for $G_{2\ 3}$ on DB_1) is about 68.5% of the corresponding average observed cost. The latter case can be improved by refining the query classification, as we will see later.

- The statistical F-tests at the significance level $\alpha = 0.01$ show that all derived cost formulas are useful for estimating the costs of queries in the relevant query classes.

- The statistical hypothesis tests for the Spearman's rank correlation coefficients at the significance level $\alpha = 0.01$ show that there is no strong evidence indicating the violation of equal variances assumption for all derived cost formulas after using the method of weighted least squares when needed.

- Derivations of most cost formulas require the method of weighted least squares (WLS), which implies that the error terms of the original regression model (using the regular least squares) violate the assumption of equal variances in most cases.

- The cost formula for a query class with clustered indexes (i.e., $G_{1\ 1}$, $G_{1\ 3}$, or $G_{2\ 1}$) appears better than the cost formula for the corresponding query class with non-clustered indexes (e.g., $G_{1\ 2}$, $G_{1\ 4}$, or $G_{2\ 2}$). The reason for this is that non-clustered indexes usually have diverse clustering ratios.

Although the statistical measures show that all derived cost formulas are useful for estimating query costs, this fact needs to be further substantiated by checking if the cost formulas can give acceptable estimated costs for test queries.

Test queries for different query classes, as described earlier in this section, were performed on the three component database systems. Their costs were observed. The derived cost formulas were used to give estimated costs for the test queries. The comparisons of observed and estimated costs of the test queries were made. Figures 7 $\sim$ 12 show such comparisons for some typical query classes. Experiments demonstrated that the estimated costs for the majority of test queries have relative errors within 30%. This observation further confirmed that the derived cost formulas are useful for estimating query costs.

As pointed out in previous sections, each cost formula can be further improved by refining the query classification, adding more significant explanatory variables, and/or using a better formulation.

For example, to improve the standard error of estimation for the cost formula for $G_{2\ 3}$ on DB_1, $G_{2\ 3}$ can be further divided into a number of smaller query classes. One such smaller class $G'_{2\ 3}$, for instance, contains the queries satisfying the following conditions:

- they are in $G_{2\ 3}$;
- only one (assume the left) of the two operand tables in a query has an index-usable separable conjunctive term;
- the separable conjunctive term is non-clustered-index-usable via a key; i.e., $R_i.a_n = C^{(i.a_n)}$ where $R_i.a_n$ is non-clustered-indexed.

Sample and test queries for $G'_{2\ 3}$ were generated in a similar way for $G_{2\ 3}$. The same statistical regression analysis procedure was invoked to derive a cost formula for $G'_{2\ 3}$ based on observed costs of sample queries. The derived cost formula and relevant statistical measures for $G'_{2\ 3}$ are given in Table 13. From the table, we can see that, the standard error

Table 13. Cost Formula and Statistical Measures for $G'_{2\ 3}$ on ORACLE 7.0

experimental data for $G'_{2\ 3}$	Cost Estimation Formula					
	0.196983 $+$ 0.889058e-4 $* N_{J2}$ $+$ 0.960488e-2 $* TN_{J1}$ $+$ 0.12004e-2 $* TN_{J2}$ $+$ 0.874517e-3 $* N_{J12}$ $+$ 0.474442e-2 $* RN_J$					
	coef. of multiple determ.	standard err. of estimation	average cost (sec.)	F-statistic (critical value at $\alpha = 0.01$)	Spearman's rank coefficient (critical value at $\alpha = 0.01$)	WLS ?
	0.9992	0.325	0.771e+1	0.275e+5 ($>$ 3.48)	0.064 ($<$ 0.215)	yes

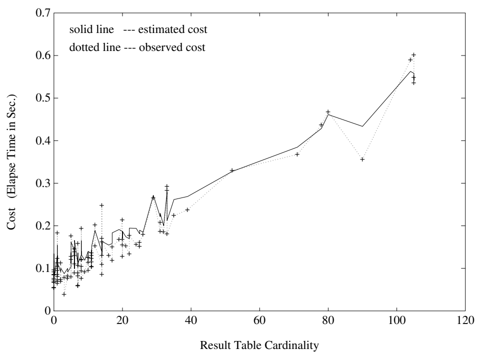

Figure 7. Observed and Estimated Costs for Test Queries in G_{11} on ORACLE 7.0

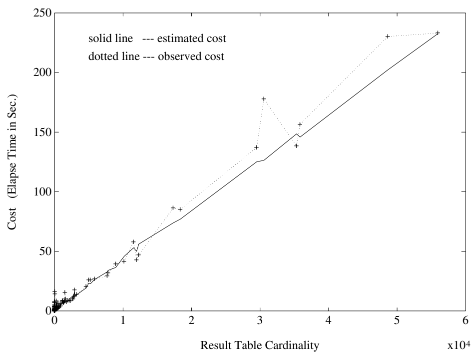

Figure 8. Observed and Estimated Costs for Test Queries in G_{21} on ORACLE 7.0

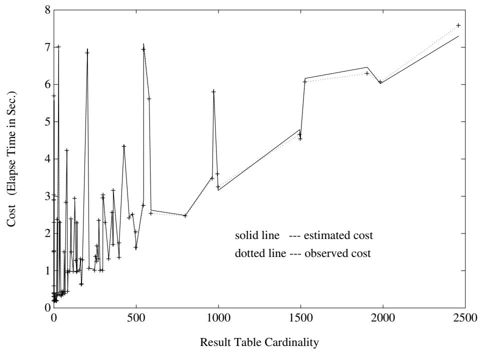

Figure 9. Observed and Estimated Costs for Test Queries in G_{15} on EMPRESS 4.6

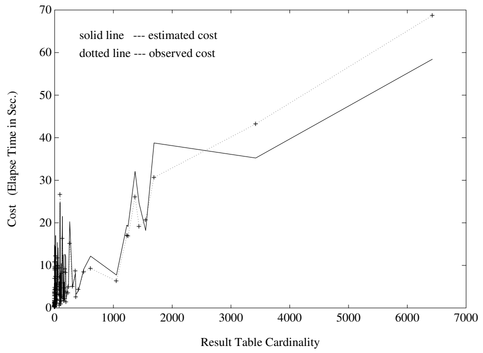

Figure 10. Observed and Estimated Costs for Test Queries in G_{22} on EMPRESS 4.6

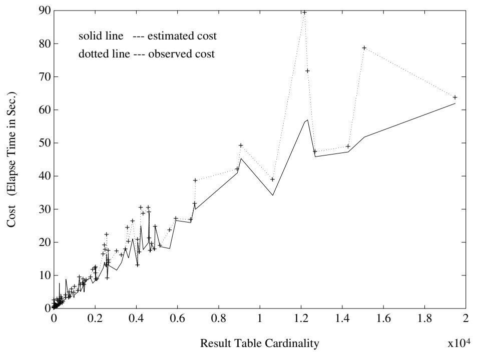

Figure 11. Observed and Estimated Costs for Test Queries in G_{14} on DB2/6000

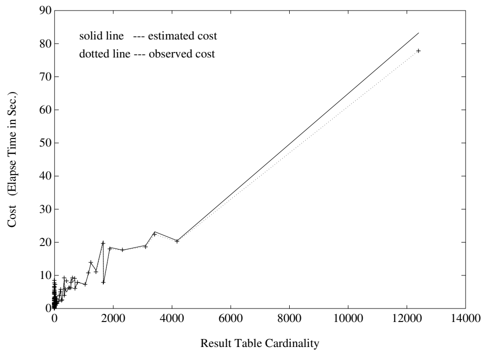

Figure 12. Observed and Estimated Costs for Test Queries in G_{23} on DB2/6000

of estimation has been improved to 4.2% of the average observed cost, and the coefficient of total determination has also been improved significantly.

One observation was noticed during our experiments. That is, small-cost queries often have worse estimated costs than large-cost queries. This observation coincides with Du *et al.*'s observation for their calibration method [7]. The reason for this phenomenon is that (1) a cost formula is usually dominated by large costs used to derive it, while the small costs may not follow the same formula because different buffering and processing strategies may be used for the small-cost queries; (2) a small cost can be greatly affected by some contention factors, such as available buffer space and the number of concurrent processes; (3) initialization costs, distribution of data over a disk space and some other factors, which may not be important for large-cost queries, could have major impact on the costs of small-cost queries.

Since the causes of this problem are usually uncontrollable and related to implementation details of the underlying component database system, it is hard to completely solve this problem at the global level in an MDBS. However, this problem could be mitigated by (a) refining the query classification according to the sizes of result tables; and/or (b) performing a sample query multiple times and using the average of observed costs to derive a cost formula; and/or (c) including in the cost formula more explanatory variables if available, such as buffer sizes, and distributions of data in a disk space.

Fortunately, estimating the costs of small-cost queries is not as important as estimating the costs of large-cost queries in query optimization because it is more important to identify large-cost queries so that ‘‘bad’’ execution plans could be avoided.

Note that experimental results may change from one experiment to another, depending on the system environment. Such changes may be relatively small for large-cost queries, but may be significant for small-cost queries. Therefore, unless additional explanatory variables reflecting the changing environment are included, the cost formulas should be used in an environment similar to where they were derived.

7. Conclusion

Local autonomy, which is the key feature of a multidatabase system, poses many new challenges for global query optimization in such a system. A crucial challenge is that some local information needed for global query optimization, such as local cost parameters, may not be available at the global level. To perform global query optimization, methods to derive/estimate the cost parameters of an autonomous component database system at the global level are required.

In this paper, a new query sampling method to estimate local cost parameters in an MDBS has been proposed. The idea is to: (1) group component queries into more homogeneous classes so that the costs of queries in each class can be satisfactorily estimated by the same formula; (2) draw a sample of queries from each query class and perform them on the relevant component DBS; (3) use the observed costs of sample queries to drive a cost estimation formula for each query class by multiple regression. To estimate the cost of a query, we first identify the class to which the query belongs, then apply the corresponding cost formula to calculate an estimate. The issues related to this method are discussed in this paper.

To classify queries, three types of information available at the global level of an MDBS, i.e., characteristics of queries, characteristics of operand tables, and characteristics of underlying component DBMSs, are utilized. A suggested query classification principle is to put queries that likely employ the same access method into the same class. The reason for doing this is that the queries employing the same access method usually follow the same performance pattern; that is, their costs can be estimated well by the same formula. To achieve this goal, a number of query classification rules that are based on the common policies used in many DBMSs for choosing access methods are introduced. A query classification obtained by applying these rules is described. A query classification can be further refined if more information about the underlying component DBS is available.

To draw sample queries from a query class, a two-phase sampling approach is suggested. The idea is to use some known knowledge to reduce a query class to a smaller set of representative queries first, then draw a set of random queries from the representative set as a sample for the given query class. The sampling method used in the second phase is a mixture of simple random sampling, stratified sampling and cluster sampling. Several principles for sampling queries from a query class are suggested. Sampling procedures for different query classes are presented. It is shown that all the presented sampling procedures can guarantee to produce a sample with any required minimum sample size.

A set of explanatory variables that can be included in a cost formula for a query class are identified. A cost model (formula) is developed for each query class. A mixed forward and backward procedure is presented to automatically select significant variables in a cost formula. Multiple regression is used to estimate the coefficients of a cost formula. Measures to handle outliers, multicollinearity, and unequal variances are suggested. In particular, the method of weighted least squares is utilized to remedy the possible violation of equal variances assumption for a multiple regression model. The significance of a regression cost model is tested by using the standard error of estimation, the coefficient of total determination, F-test, and test queries.

The processes for query classification, query sampling, and cost formula derivation may need to be iterated several times before satisfactory cost formulas are achieved. A complete statistical procedure integrating all the processes is described.

To check the feasibility of the query sampling method, experiments on three commercial DBMSs --- ORACLE 7.0, EMPRESS 4.6, and DB2/6000 1.1.0 were conducted. The experimental results demonstrated that the query sampling method is quite promising in estimating local cost parameters in an MDBS. Statistical measures indicated that the cost formulas derived by using the query sampling method are significant. These derived cost formulas indeed produced good cost estimates for the majority of test queries in the experiments. In particular, the cost estimates for large-cost queries are even better than those for small-cost queries, which is desired by query optimization.

The main advantages of the query sampling method are: (1) it only uses information available at the global level in an MDBS --- no special privilege is required from a component database system; (2) it uses a real component database, instead of a special synthetic calibrating database, to derive cost parameters, so the derived parameters reflect the real environment in practice; (3) the cost formula for a query can be easily identified by recognizing the query class to which the query belongs; (4) the significant variables in a cost formula are automatically selected, and insignificant variables are not included; (5) the derived local cost parameters can be stored in the multidatabase catalog instead of built into the global query optimizer, so a new component DBS can easily be added into the MDBS; (6) a cost formula can be dynamically improved to reflect a changing environment by periodically re-performing sample queries; (7) the method is robust; that is, any reasonable query classification and sampling can produce usable cost formulas.

The work reported in this paper is only the beginning of more research that needs to be done in order to completely solve the problem of estimating local cost parameters in an MDBS. Many issues need to be further investigated in the future, such as how to use the query sampling method to handle queries involving aggregation functions and views, how to incorporate system contention factors into a cost model, how to make use of application knowledge to derive cost formulas that fit better to practically-used queries, how to refine a query classification by analyzing observed costs of queries and recognizing their performance patterns, how to efficiently make use of observed costs of user queries to improve the cost formulas, how to extend the method to directly handle non-relational local data models such as object-oriented model.

In summary, the problem of estimating local cost parameters in an MDBS is challenging; the query sampling method proposed in this paper introduces a promising approach to solve the problem; and many interesting issues remain to be resolved in the future.

Acknowledgments

We would like to thank Grant E. Weddell, Frank W. Tompa, Amit Sheth, Kenneth Salem, Patrick Martin, Jacob Slonim, M. Tamer Ozsu, Frank Olken, Weimin Du, Witold Litwin, Neil Cobrun, and Yigal Gerchak for their insightful suggestions and comments. We are grateful to Lauri Brown and Wendy Powley for their help in setting up the experimental environments. We would also like to thank the anonymous referees for their careful reading, valuable suggestions, and encouraging comments. This research was supported by the IBM Toronto Laboratory and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Notes

1. Any query has at least one conjunct $R_i.a_n\ \vartheta\ C^{(i.a_n)}$ ($\vartheta \in \{<, \leq, \geq, >, =, \neq, nil\}$). The '$true$' predicate $R_i.a_n\ nil\ C^{(i.a_n)}$ is considered only if there is no other conjunct of such a form in the query. As said, only $<,\ >,\ =,\ \neq$ are considered in the representative queries.
2. Note that the auxiliary conjunct in a representative query may also be a key conjunct. To avoid confusion, we always refer to the first conjunct when we say the key conjunct of a representative query.
3. Notations $\widetilde{C}^{(1.a_1)}$ and $C_{\underset{i}{\sim}}^{(1.a_1)}$ are the same. Both denote a random constant chosen from the domain of $R_1.a_1$.
4. Note $\frac{|\Delta'_U|}{|\Delta'_U|+4*|\Delta''_U|}$ is the fraction of representative key conjunct types induced by the columns in Δ'_U among all representative key conjunct types induced by the columns in Δ_U.
5. For simplicity, we assume that $\Delta_{JA}|_{(R_i,R_j)}$ has sufficient pairs.
6. We assume that the tuples in a table have a fixed length. Otherwise, the average tuple length could be utilized.
7. If there are more than one index-usable key predicates (conjuncts) for the unary query, the average of their selectivities could be used.
8. The physical size of an operand table can be more accurately estimated by $(N_U + d_1) * L_U * d_2$, where the constants d_1 and d_2 reflect some overhead such as page overhead and free space [5]. Since the constants d_1 and d_2 are applied to all sample data, they can be omitted. Estimating the physical size of a result table is similar.
9. The simple correlation coefficient between two variables indicates the degree of the linear relationship between the two variables [17].
10. The standard error of estimation is an indication of the accuracy of estimation [17].
11. The coefficient of multiple determination measures the proportion of variability in the response variable explained by the explanatory variables [17].

References

1. Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of the ACM SIGMOD Conference*, pages 135--142, 1988.
2. M. W. Bright, A. R. Hurson, and S. H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3):50--59, Mar. 1992.
3. S. Chatterjee and B. Price. *Regression Analysis by Example, 2nd Ed.* John Wiley & Sons, Inc., 1991.
4. W. G. Cochran. *Sampling Techniques*. John Wiley & Sons, Inc., 1977.
5. IBM Corp. Database 2 OS/2 guide. User manual, IBM Canada Ltd. Lab., North York, Canada, 1993.
6. U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. Soft. Eng.*, SE-10(6):628--644, Nov. 1984.
7. W. Du, R. Krishnamurthy, and M. C. Shan. Query optimization in heterogeneous DBMS. In *Proceedings of VLDB*, pages 277--91, 1992.
8. W. C. Hou et al. Error-constrained COUNT query evaluation in relational databases. In *Proceedings of SIGMOD*, pages 278--87, 1991.
9. M. Jarke and J. Koch. Query optimization in database systems. *Computing Surveys*, 16(2):111--152, June 1984.
10. R. J. Lipton and J. F. Naughton. Practical selectivity estimation through adaptive sampling. In *Proceedings of SIGMOD*, pages 1--11, 1990.
11. W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267--293, Sept. 1990.
12. H. Lu, B.-C. Ooi, and C.-H. Goh. On global multidatabase query optimization. *SIGMOD Record*, 21(4):6--11, Dec. 1992.
13. H. Lu and M.-C. Shan. On global query optimization in multidatabase systems. In *2nd Int'l workshop on Research Issues on Data Eng.*, page 217, Tempe, Arizona, USA, 1992.
14. M. Muralikrishna and D. J. DeWitt. Equi-Depth histograms for estimating selectivity factors for multi-Dimensional queries. In *Proceedings of SIGMOD*, pages 28--36, 1988.
15. J. Neter, W. Wasserman, and M. H. Kutner. *Applied Linear Statistical Models, 3rd Ed.* Richard D. Irwin, Inc., 1990.
16. F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proceedings of 12th VLDB*, pages 160--9, 1986.
17. R. C. Pfaffenberger and J. H. Patterson. *Statistical Methods for Business and Economics*. Richard D. Irwin, Inc., 1987.
18. P. G. Selinger et al. Access path selection in relational database management systems. In *Proceedings of ACM SIGMOD*, pages 23--34, 1979.
19. G. P. Shapiro and C. Connel. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of SIGMOD*, pages 256--76, 1984.
20. A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183--236, Sept. 1990.
21. N. Wang, P. Zhou, Qiang Zhu, et al. NITDB : A multi-user relational DBMS for microcomputers. *Chinese Computer Journal*, 10(8):477--84, Aug. 1987.
22. N. Wang and Qiang Zhu. Query processing in a relational micro-DBMS with multiple optimization strategies. *Computer Research and Development*, 23(9):24--30, Sept. 1986.
23. N. Wang and Qiang Zhu. Functionality and implementation techniques of the relational DBMS: NITDB. *Software Industry*, 5(9):28--37, Sept. 1988.
24. Qiang Zhu. Query optimization in multidatabase systems. In *Proceedings of the 1992 IBM CAS Conference, vol.II*, pages 111--27, Toronto, Canada, Nov. 1992.
25. Qiang Zhu. An integrated method of estimating selectivities in a multidatabase system. In *Proceedings of the 1993 IBM CAS Conference*, pages 832--47, Toronto, Canada, Oct. 1993.
26. Qiang Zhu and P.-A. Larson. A fuzzy query optimization approach for multidatabase systems. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 5(6):701--22, 1997.
27. Qiang Zhu and P.-A. Larson. A query sampling method for estimating local cost parameters in a multidatabase system. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, pages 144--53, Houston, Texas, Feb. 1994.
28. Qiang Zhu and P.-A. Larson. Establishing a fuzzy cost model for query optimization in a multidatabase system. In *Proceedings of the 27th IEEE/ACM Hawaii International Conference on System Sciences*, pages 263--72, Maui, Hawaii, Jan. 1994.