Query Optimization in Heterogeneous DBMS

Weimin Du Ravi Krishnamurthy Ming-Chien Shan
HP Labs, Mailstop 3U-4, P.O. Box 10490, Palo Alto, 94303-0969
email: lastname@hpl.hp.com

Abstract

We propose a query optimization strategy for heterogeneous DBMSs that extends the traditional optimizer strategy widely used in commercial DBMSs to allow execution of queries over both known (i.e., proprietary) DBMSs and foreign vendor DBMSs that conform to some standard such as providing the usual relational database statistics. We assume that participating DBMSs are autonomous and may not be able, even if willing, to provide the cost model parameters. The novelty of the strategy is to deduce the necessary information by calibrating a given DBMS. As the calibration has to be done as a user, not as a system administrator, it poses unpredictability problems such as inferring the access methods used by the DBMS, idiosyncrasies of the storage subsystem and coincidental clustering of data. In this paper we propose a calibrating database which is synthetically created so as to make the process of deducing the cost model coefficients reasonably devoid of the unpredictability problems. Using this procedure, we calibrate three commercial DBMSs, namely Allbase, DB2, and Informix, and observe that in 80% of the cases the estimate is quite accurate.

1 Motivation

Heterogeneity of databases and database management systems (DBMSs) have gained its due recognition as a result of the advent of open systems. Typically this heterogeneity may result from semantic discrepancies in the data, multiple data models, different systems, etc. All of these are consequences of the need for independent database systems to interoperate while preserving their autonomy. Lots of research has been done in the context of integration and interoperability, where the center of focus has been in the semantics of translation and integration[HWKSP1, HWKSP2]. Very few papers have been published on the problem of query processing [Da85] and even less on query optimization in the context of heterogeneous DBMSs, referred to as *heterogeneous query optimization, or HQO*.

In [LS92] some important issues concerning HQO problem were enumerated. In this paper, we address the problem of optimization of queries over heterogeneous DBMSs, where the degree of heterogeneity, with respect to the optimizer, may vary over the participating systems. These participating DBMSs are typically supplied by multiple database vendors and can be classified, in relationship to the integrating DBMS (referred to as HDBMS), into the following three categories:

- *Proprietary DBMS:* The participating DBMS is called a proprietary DBMS if it is from the same vendor as the HDBMS. A vendor would naturally like to have an optimizer that is at least as good as its DBMS product. This necessitates that the optimizer is at least as capable as their participating DBMS optimizer and uses all the relevant information on cost functions and database statistics[1]. The HQO problem in the context of proprietary DBMS is quite similar to the distributed query optimization problem.
- *Conforming DBMS:* The participating DBMS is called a conforming DBMS if it is from a foreign vendor and the DBMS is capable of providing some important database statistics but incapable of divulging the cost functions either due to the lack of such abstractions in the system or due to such information being proprietary.
- *Non-Conforming DBMS:* The participating foreign vendor DBMS is incapable of divulging either the database statistics or the cost functions.

[1]In fact, any non-proprietary DBMS that is capable of providing the necessary information (i.e., cost functions and database statistics) for customizing the optimizer can be accommodated in this category.

Proceedings of the 18th VLDB Conference Vancouver, British Columbia, Canada, 1992.

This would be the case when supporting a data server that exports a set of functions. Therefore, the system has to be viewed as a black box whose only characteristics known to the optimizer are those observable by the execution of queries.

The HDBMS query optimizer should be capable of not only providing the expected level of optimization when the participating systems are all proprietary but also degrading gracefully in the presence of either conforming or non-conforming DBMSs. Thus, it is imperative for the optimizer to view these systems in a seamlessly integrated fashion. We present such an approach for the optimizer that not only achieves this goal but also is presented as an extension to the widely used architecture for the optimizer proposed in [Sell79].

In HDBMSs, the execution space must be extended to handle global queries across DBMSs. This can be done by simply allowing new join methods across DBMSs. Therefore, the execution space and search strategy used in commercial DBMSs can be used in HDBMS if only a cost model is made available for all categories of DBMSs. In short, the crux of the HQO problem is in deriving the cost model for autonomous DBMSs. This involves calibrating a given relational DBMS and deducing the cost coefficients of the cost formulae.

As the calibration has to be done as a user to the DBMS, the association of cause and (observed) effect poses unpredictability problems. Thus, such a proposal should be able to answer questions such as, is the system actually using index scan for the inner loop of the join, is the observed value not distorted by some idiosyncrasies of the storage management of data, and is the observed value due to per-chance clustering of data.

The novelty of our approach is in the design of the synthetic database and the properties that can be derived for both the database and the DBMS. Prior benchmarking efforts [BDT83] has relied on generating the database in a probabilistic fashion. In contrast we provide a deterministic generation algorithm using which formal properties of that relation can be stated. These properties ensure the avoidance of unpredictability problems. We argue that some of these properties cannot be attained using the traditional benchmarking approach. Another advantage of the synthetic database is that a million tuple database can be generated in minutes whereas the technique used in [BDT83] would take significantly more time.

A calibrating procedure has been presented that has been applied to three commercial systems — namely Allbase, DB2, and Informix. The validation results indicate that this approach is reasonably accurate to the extent that estimated values of majority of the validation queries were within 20% error of the observed values. All the queries that had error in excess of 20% were found to be of a particular type. Being the first attempt to calibrate the system using an arms-length procedure, this is not only novel but also promising.

In section 2 we review the traditional optimizer technology and conclude that the crux of the problem is to design cost models for proprietary and conforming DBMSs such that they can be used by the heterogeneous query optimizer. Section 3 outlines an extension of the traditional cost model for heterogeneous environment. Section 4 deals with the problem of calibrating an autonomous DBMS. Here a logical cost model is presented that is devoid of physical information such as page sizes, number of I/Os, etc. Using this cost model, a calibrating procedure using the calibrating database is presented. Then we present practical calibrations of Allbase, DB2, and Informix and show that the estimated values are reasonably accurate. Finally we conclude that such an approach is practical based on the following widely held maxim for the traditional query optimizer: "It is more important to avoid the worst execution than to obtain the best execution"[KBZ86].

This work has been done in the context of Pegasus project at HP Labs.

2 Optimizer Overview

A heterogeneous DBMS, HDBMS, can be viewed as a DBMS that has a set of participating DBMSs and provides both interoperability of DBMSs as well as integration of data. In this paper we assume that participating DBMSs are all relational and for the sake of simplicity assume that all data are schematically and representationally compatible; i.e., there is no integration problem. In this sense, the HDBMS provides a point for querying all the participating DBMSs and provides database transparency (i.e., the user needs not to know where data reside and how queries are decomposed). Therefore, the HDBMS is relegated the responsibility for decomposing and executing the query over the participating DBMSs. This is the motivating need for a query optimizer for HDBMSs. Without loss of generality, we assume that the query is a conjunctive relational query.

2.1 Traditional Optimizer Revisited

The optimization of relational queries can be described abstractly as follows:

Given a query Q, an execution space E, and a cost function C defined over E, find an execution e in E_Q that is of minimum cost, where E_Q is the subet of E that computes Q.

$$min_{e \in E_Q} C(e)$$

Any solution to the above problem can be characterized by choosing: 1) an execution model and, therefore, an execution space; 2) a cost model; and 3) a search strategy.

The execution model encodes the decisions regarding the ordering of the joins, join methods, access methods, materialization strategy, etc. The cost model computes the execution cost. The search strategy is used to enumerate the search space, while the minimum cost execution is being discovered. These three choices are not independent; the choice of one can affect the others. For example, if a linear cost model is used, as in [KBZ86], then the search strategy can enumerate a quadratic space; on the other hand, an exponential space is enumerated if a general cost model is used, as in the case of commercial database management systems. Our discussion in this paper does not depend on either the execution space (except for the inflection mentioned below) or the search strategy employed. However, it does depend on how the cost model is used in the optimization algorithm. Therefore, we make the following assumptions.

The execution space used in [Sell79] can be abstractly modeled as the set of all join orderings (i.e., all permutations of relations) in which each relation is annotated with access/join methods and other such inflections to the execution. These and other inflections to the tradional execution space is formalized in the next subsection.

We assume the exhaustive search as the search strategy over the execution space, which has been widely used in many commercial optimizers. This space of executions is searched by enumerating the permutations and for each permutation choosing an optimal annotation of join methods and access methods based on the cost model. The minimum cost execution is that permutation with the least cost.

The traditional cost model uses the description of the relations, e.g., cardinality and selectivity, to compute the cost. Observe that the operands for these operations such as select and join may be intermediate relations, whose descriptions must be computed. Such a descriptor encodes all the information about the relation that is needed for the cost functions.

Let the set of all descriptors of relations be $\mathcal{D}$, and let $\mathcal{I}$ be the set of cost values denoted by integers. We are interested in two functions for each operation, σ, such as join, select, project etc. These functions for a binary operation σ are:

$COST_\sigma : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{I}$ *and* $DESC_\sigma : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$.

Intuitively, the $COST_\sigma$ function computes the cost of applying the binary operation σ to two relations, and $DESC_\sigma$ gives a descriptor for the resulting relation. The functions for the unary operators are similarly defined. In this paper we shall be mainly concerned with the $COST$ function and assume the traditional definitions for the $DESC$ function given in [Sell79]. This has the information such as the cardinality of the relation, number of distinct values for each column, number of pages, index information, etc.

2.2 Execution Model for HDBMSs

Here we outline the space of executions allowed by the HDBMS. Intuitively, an execution plan for a query can be viewed as a plan for a centralized system wherein some of the joins are across DBMSs. These joins can be viewed as alternate join methods. Here we formally define a plan for a traditional DBMS and then extend it by allowing these new join methods.

A plan for a query, in most commercial DBMSs, denotes the order of joins and the join method used for each join and the access plan for each argument of the join. Besides this, we assume the usual information needed to denote the sideway information passing (also known as the binding information).

Formally, a plan for a given conjunctive query is expressed using a normal form defined in [CGK90] called *Chomsky Normal Form* or CNF. A CNF program is a Datalog program in which all rules have at most two predicates in the body of the rule. Consider a conjunctive query on k relations. This can be viewed as a rule in Datalog with k predicates in the body. An equivalent CNF program can be defined using $k-1$ rules each of which has exactly two predicates in the body. This is exemplified below.

$p(X,Y) \leftarrow r_1(X,X_1), r_2(X_1,X_2), r_3(X_2,X_3), r_4(X_3,Y).$

has the following equivalent CNF program:

$p(X,Y) \leftarrow r_{13}(X,X_3), r_4(X_3,Y).$
$r_{13}(X,X_3) \leftarrow r_{12}(X,X_2), r_3(X_2,X_3).$
$r_{12}(X,X_2) \leftarrow r_1(X,X_1), r_2(X_1,X_2).$

In the above CNF program we give the added restriction that the execution is left to right in the body of the rule. As a result, the sideway information passing through binding is from r_{13} to r_4 in the first rule. Further note that the CNF program completely denotes the ordering of joins of the relations. Therefore, r_1 and r_2 are joined before the join with r_3. Note that the above CNF program can represent all type of bushy join trees; in contrast most commercial DBMSs allow only left deep join trees. For simplicity, we make the assumption that all CNF programs are left-deep (i.e., only the first predicate in the body can be a derived predicate) and thus omit other CNF programs from consideration.

For each join in the CNF program, a join method is assigned; one join method is associated with the body of each rule. Two join methods are considered: nested loop (NL) and ordered merge (OM) such as sort merge or hash merge. Further, each predicate is annotated with an access method; i.e., at most[2] two access methods per rule. The following are four access methods we considered:

- sequential scan (SS): access and test all tuples in the relation;
- index-only scan (IO): access and test index only;
- clustered index scan (CI): access and test clustered index to find qulifying tuples and access the data page to get the tuple; and
- unclustered index scan (UI): same as CI, but for unclustered index.

This list of methods reflect the observations in many commercial DBMSs such as differentiating accessing index page only versus accessing both index and data pages. Access to clustered and unclustered index pages is not differentiated.

Even though all access methods can be used with all join methods, some combinations do not make sense. Nevertheless, we assume that the optimizer can avoid these cases by assigning a very large cost. Note that the list of join methods and access methods can easily be extended and the cost formulae specified in a similar manner. For the discussion in most of this paper, this list of methods would suffice. We shall discuss in specific context some variations to these access methods by inflecting them with techniques such as preselect (also known as prefetch), create index, etc.

As mentioned before, the list of join methods is increased by a new method called *remote join* that is capable of executing a join across two DBMSs. This may be done by shipping the data directly to the other DBMS or to the HDBMS which in turn coordinates with the other DBMS to compute the join. Obviously, there is a host of variation in achieving this remote join and distributed DBMS literature has a wide variety of them documented [CP84]. For expository simplicity, we assume that some such remote joins are chosen. For each such remote join, a cost function is associated and the cost of the complete execution is computed in the traditional manner. Even though the specific choices for the remote joins and the cost model used for estimating the cost of remote join are important for a HDBMS optimizer, we can omit this aspect of the problem in the paper without loss of generality.

In summary, we view the optimizer as one that searches a large space of executions and finds the minimum cost execution plan. As execution space and search strategy largely remain unchanged, we need only describe the cost model for each category of DBMSs in order to describe the optimizer for HDBMSs. In particular, we shall concentrate on the cost model for select and join operations in the context of a single DBMS, using which joins across DBMSs can be computed based on the remote join cost model. The cost models for proprietary and conforming DBMS are the topics of the next two sections.

3 Cost Model for Proprietary DBMSs

As mentioned before, the cost model for a query over multiple proprietary DBMSs must be comparably capable to the cost model used in the DBMS itself. This requires that the cost model knows the internal details of the participating DBMS. For a proprietary DBMS, this is possible. In this section, we outline the cost model at a very high level. Our purpose is not to present the model details per se but to observe the use of physical parameters such as prefetch, buffering, page size, number of instructions to lock a page and many other such implementation dependent characteristics.

Typically the cost model estimates the cost in terms of time; in particular the minimum elapsed time that can occur. This estimated time usually does not predict any device busy conditions, which in reality would increase the elapsed time. We use this notion of time as the metric of cost. This is usually justified on the ground that minimizing this elapsed time has the effect of minimizing the total work and thereby 'maximizing' the throughput.

In HDBMSs, elapsed time can be estimated by estimating three components:

[2]Note that in some cases such as the outer loop (of a nested loop join) that is being pipelined from another join, annotation of an access method does not make sense.

- CPU time incurred in both the participating DBMSs and the HDBMS.
- I/O time incurred in both the participating DBMSs and the HDBMS.
- Communication time between the HDBMS and the participating DBMSs.

Traditional centralized DBMSs included only CPU and I/O time in their estimate. We propose to use a similar estimate for both these components. For each of the join methods and access methods allowed, a cost formula is associated.

The CPU estimate includes the time needed to do the necessary locking, to access the tuples either sequentially or using an index, to fetch the tuples, to do the necessary comparisons, to do the necessary projections, etc. The cost formulae are based on estimating the expected path length of these operations and on the specific parameters of the relations such as number of tuples, etc. Obviously, these parameters are continually changing with the improvement in both the hardware and the software. For a proprietary system these changes can be synchronized with the new versions.

The I/O time is estimated using the device characteristics, page size, prefetch capabilities, and the CPU path length required to initiate and do the I/O.

The time taken to do the necessary communication can be estimated based on the amount of data sent, packet size, communication protocol, CPU path length needed to initiate and do the communication, etc. It is assumed that the physical characteristics of the communication subsystem are known to the HDBMS and accurate cost model can be developed using these parameters.

In summary, we have described the cost model for a proprietary DBMS to be one that has complete knowledge of the internals of the participating DBMS.

4 Cost Model for Conforming DBMSs

A conforming DBMS is a relational DBMS with more or less standard query functionality. The purpose in this section is to design a cost model that would estimate the cost of a given query such that the model is based on the logical characteristics of the query. In this section, we first outline a cost model for estimating the cost of a given plan for a query. Then we describe the procedure by which to estimate the constants of the cost model as well as experimental verification of this procedure. Finally, a dynamic modulation of these constants is presented to overcome any discrepancy.

4.1 Logical Cost Model

The logical cost model views the cost on the basis of the logical execution of the query. There are two implications. First, the cost of a given query is estimated based on logical characteristics of the DBMS, the relations, and the query. Second, the cost of complex queries (e.g., nested loop joins) is estimated using primitive queries (e.g., single table queries). In this subsection, we outline such a cost model.

4.1.1 Cost Formulae

For the sake of brevity, we restrict our attention to that part of the cost model which estimates the cost of select and join operations. Formally, given two relations r_1 and r_2 with N_1 and N_2 tuples, we estimate the cost of the following two operations:

- a select operation on r_1 with selectivity S_1; and
- a join operation on r_1 and r_2 with selectivity J_{12}.

The formulae for these two operations are given in Figure 1 with the following assumptions, all of which will be relaxed in the next subsection:

- The size of the tuple is assumed to be fixed.
- There is exactly one selection/join condition on a relation.
- The entire tuple is projected in the answer.
- All attributes are integers.

Intuitively, the select cost formulae can be viewed as a sum of the following three (independent) components:

$COMP_0$: initialization cost.
$COMP_1$: cost to find qulifying tuples.
$COMP_2$: cost to process selected tuples.

The $COMP_0$ component is the cost of processing the query and setting up the scan. This is the component that is dependent on the DBMS, but independent of either the relation or the query.

In the case of sequential scan, the $COMP_1$ component consists of locking overhead, doing the 'get-next' operation, amortized I/O cost and all the other overhead incurred per tuple of the scan. Note that the fixed size of the tuple assures that the number of pages accessed is directly proportional to the number of tuples. Therefore, the cost per tuple is a constant. In the case

The selection cost formulae are categorized as follows:

- **Sequential Scan:**
  $CS_{ss} = CS0_{ss} + ((CS1_{ss}^{io} + CS1_{ss}^{cpu}) * N_1) +$
  $(CS2_{ss} * N_1 * S_1)$
- **Index-Only Scan:**
  $CS_{is} = CS0_{is} + CS1_{is} + (CS2_{is} * N_1 * S_1)$
- **Clustered Index Scan:**
  $(CS_{ci} = CS0_{ci} + CS1_{ci} + (CS2_{ci} * N_1 * S_1)$
- **Unclustered Index Scan:**
  $CS_{ui} = CS0_{ui} + CS1_{ui} + (CS2_{ui} * N_1 * S_1)$

where,

$CS0_{xx}$	(i.e., $CS0_{ss}, CS0_{is}, CS0_{ci}$ and $CS0_{ui}$) The initialization cost for selects;
$CS1_{ss}^{io}$	The amortized I/O cost of fetching each tuple of the relation, irrespective of whether the tuple is selected or not.
$CS1_{ss}^{cpu}$	The CPU cost of processing each tuple of the relation (e.g., checking selection conditions).
$CS1_{is}$	(similarly, $CS1_{ci}$ and $CS1_{ui}$) The cost of initial index lookup.
$CS2_{ss}$	The cost of processing a result tuple for sequential scan.
$CS2_{is}$	(similarly, $CS2_{ci}$ and $CS2_{ui}$) The cost of processing each tuple selected by an index. This includes the I/O cost to fetch tuple if necessary.

The join cost formulae are categorized as follows:

- **Nested Loop:** (if sequential scan on r_2)
  $CJ_{nl} = CS_{xx}(r_1) + CS0_{ss}(r_2) + CS1_{ss}^{io}(r_2) +$
  $(N_1 * S_1 * (CS_{ss}(r_2) - CS0_{ss}(r_2) - CS1_{ss}^{cpu}(r_2)))$
- **Nested Loop:** (if index scan on r_2)
  $CJ_{nl} = CS_{xx}(r_1) + CS0_{xx}(r_2) +$
  $(N_1 * S_1 * (CS_{xx}(r_2) - CS0_{xx}(r_2))$
- **Ordered Merge:**
  $CJ_{sm} = CJ1_{or}(r_1) + CJ1_{or}(r_2) + CS_{ss}(r_1) +$
  $CS_{ss}(r_2) + CJ2_{mg} * N_1 * N_2 * J_{12}$

where,

$CJ1_{or}$	The cost of ordering (e.g., sorting) a relation. Note that it may be zero if there is an index on the joining column.
$CJ2_{mg}$	The cost of merging join tuples.

Figure 1: Cost Formulae for Select and Join

of index scans, $COMP_1$ is the initial index look up cost. Note that in all the three cases of index scans, we assume that $COMP_1$ is independent of the relation. This is obviously not true as the number of levels in the index tree is dependent on the size of the relation. But this number is not likely to differ by a lot due to high fanout of a typical B-tree or any other indexing mechanism. Thus we believe this independence assumption is reasonable.

The $COMP_2$ component is the cost of processing each of the selected tuples. This component cost is also likely to be a constant because of the selection and projection assumption made earlier.

The join formulae are straightforward derivation from the nested loop and ordered merge algorithms. Note that they are composed of selection cost formulae with certain adjustments to handle operations (e.g., sorting and merging) and problems (e.g., buffering effect) special to join operations.

For example, the cost of accessing a relation in the inner most loop is modeled in the same manner as that of accessing a relation in the outer most loop. This is obviously not the case if the inner most relations are buffered to avoid the I/O. This uniformity was experimentally found to be quite acceptable for index scans. If a sequential scan is used in the inner loop of the join, the I/O cost may only incur once (for small tables) due to buffering whereas the table look up is done once for each tuple in the outer loop. In order to handle this, $COMP_1$ is broken into two parts in the formulae for sequential scans. These two parts represent the I/O and the CPU costs of 'get-next' operation.

4.1.2 Relaxing Assumptions

Let us now relax the assumptions made in the previous subsection. As the size of the tuple is varied, the constants will be affected. In particular only the constant $CS1_{ss}$ ($= CS1_{ss}^{io} + CS1_{ss}^{cpu}$) is expected to be affected and is likely to increase linearly with the size of the tuple. All the other constants are component costs for selected tuples and thus are not likely to be affected. But these constants (i.e., $CS2_{xx}$'s) are affected by the number of selection and projection clauses. In order to take these into account, we redefine the above constants to be functions with the following definitions:

- $CS1_{ss}$ is a linear function on the size of the tuple in the relation.
- $CS2_{xx}$'s are linear functions on the number of selection and projection clauses.

As these are no longer constants we refer to them as coefficients.

Assuming that the checking is terminated by the first failure, the expected number of select condition checked is bounded by 1.5 independent of the number of selection conditions. This was formally argued in [WK90]. Further more, the experiments show that the costs of checking selection conditions and projecting attributes are negligible comparing to other costs, e.g., I/O (see Section 4.3).

Finally, we relax the assumption that all attributes are integers by requiring one set of cost formulae for each data type in the DBMS. The respective coefficients would capture the processing, indexing, paging characteristics of that data type. As this is orthogonal to the rest of the discussion, we present the rest of the paper for just one data type.

The fundamental basis for the above cost model is that the cost can be computed based on factoring the costs on a per tuple basis in the logical processing of the query. Indeed, the coefficients $CS0_{xx}, CS1_{xx}$, and $CS2_{xx}$ are all composite cost including CPU, I/O, and any other factors that may be of interest such as the cost of connection. In this sense, the cost formulae are logical versus the cost formulae used in most commercial DBMSs which estimate based on physical parameters. The above formulae are structurally very similar to the ones that were used in the commercial DBMSs. There are three major differences.

- The traditional formulae assumed that the coefficients are constants. Here we view them as functions.
- The value associated to these coefficients are composite cost of CPU, I/O, etc.; whereas these costs were separately computed in the formulae used in most commercial DBMSs.
- The value associated to the coefficients also reflect other system dependent factors such as hardware speed and operating system and DBMS configuration, etc.

The above basis for computing the cost is obviously approximations of the more involved formulae used by the commercial DBMSs. But the important question is whether this approximations are significant to affect the optimizer decision? It is our claim that the above cost model will sufficiently model the behavior of the execution. We shall confirm this claim by experimenting with three commercial DBMSs.

4.2 Calibrating Database and Procedure

The purpose of the calibrating database is to use it to calibrate the coefficients in the cost formulae for any given relational DBMS. Our approach is to construct a synthetic database and query it. Cost metric values (e.g., elapsed time) for the queries are observed to deduce the coefficients. Note that there are no hooks assumed in the system and therefore the system cannot be instrumented to measure the constants (e.g., number of I/O issued) of the traditional cost model. Further, the construction of the database, posing of the query, and the observations are to be done as a user to this 'black-box' DBMS. This poses the following two major predicatability problems:

- the problem of predicting how the system will execute (e.g, use index or sequntial scan, use nested loop or sort merge) a given query;
- the problem of eliminating the effect of data placement, pagination and other storage implementation factors that can potentially distort the observations and thus lead to unpredictable behavior.

In order to deduce the coefficients in the cost formulae, it is imperative that the query and the database are set up such that the resulting execution is predictable; i.e., the optimizer will choose the predicted access and join methods. Even if the execution is predicatable, it should be free from the above mentioned distortion or else the determination of the cause for the observed effect is not possible. For example, if all the tuples having a particular value for an attribute just happened to be in a single page then the observed value can be misleading.

In this subsection we set up a database and a set of queries that are free from the above two problems and show that the coefficients of the cost formulae can be deduced.

4.2.1 Calibrating Database

For any integer n, let R_n be a relation of seven columns containing 2^n tuples. The seven attributes have the following characteristics:

C_1: integer $[0, n]$, indexed, clustered.
C_2: integer $[0, 2^n - 1]$, indexed, de facto clustered but not specified as such to DBMS.
C_3: integer $[0, n]$, indexed, unclustered.
C_4: integer $[0, n]$, no index.
C_5: integer $[0, 2^n - 1]$, indexed, unclustered.
C_6: integer $[0, 2^n - 1]$, no index.

C_7: a long character string to meet the size of the tuple requirement.

The values in these attributes are defined below. Even though the relation is a set and as such unordered, we can conceptually view it as a matrix. Thus, the i^{th} tuple in R_n is defined as follows:

- $C_1[n,i] = bell(i)$, where $bell(0) = 0, bell(i) = k$ such that $\sum_{j=0}^{k-1} f(j) < i+1 \le \sum_{j=0}^{k} f(j)$ and

$$f(j) = \begin{cases} 1, & \text{if } j = 0; \\ 2, & \text{if } j = 1 \text{ and } n \text{ is even}; \\ 4, & \text{if } j = 1 \text{ and } n \text{ is odd}; \\ 4 * f(j-1), & \text{if } j \le \lfloor \frac{n}{2} \rfloor; \\ f(j-1)/4, & \text{if } j > \lfloor \frac{n}{2} + 1 \rfloor; \\ f(j-1)/2, & \text{if } j = \lfloor \frac{n}{2} + 1 \rfloor. \end{cases}$$

- $C_2[n,i] = i$.
- $C_3[n,i] = mf[j]$ such that $i \bmod 2^{j+1} = 2^j$ where $mf[j] = \lfloor \frac{n}{2} \rfloor + (-1)^{j+1} \lfloor \frac{i+1}{2} \rfloor$.
- $C_4[n,i] = C_3[n,i]$.
- $C_5[n,i] = 2^{n-k} + j$ such that $i = 2^{k-1} * (1 + 2 * j)$ and $C_5[n,0] = 0$.
- $C_6[n,i] = C_5[n,i]$.

The value for the seventh attribute is a padding field and can be set to anything and therefore, for convenience, omitted from the rest of the discussion. Figure 4 has the relation R_4 tabulated.

The multicolumn key for the relation is (C_1,C_2). This relation is indexed on this key, in ascending order, with C_1 being the major key and C_2 being the minor key. This index is clustered in the sense that the values of the major key (i.e., C_1) are clustered. In general the values of C_2 may not be clustered. As it is clear from the construction of the relation, the values of the minor key (i.e., C_2) are also clustered. In fact, the values in C_2 are unique and have 2^n values in the range $[0, 2^n - 1]$ and therefore these values can also be in ascending order. So, in some sense, we can view this column, C_2, as a sequence number for the rows. We shall refer to this C_2 value as the row index. The need for multicolumn key and index is so that the tuples are ordered in the disk pages based on C_2 and the system is informed that C_1 has a clustered index. This could be achieved by inserting the tuples in that order as long as the index creation in DBMS is stable, which most DBMSs do satisfy[3].

Note that the database can be generated by a procedure that evaluates the above formulae. Therefore, it is possible to generate a million tuple database in minutes. In contrast the database generated using traditional benchmarking technique [BDT83] would take significantly more time.

[3]In fact this was used in calibration of systems presented in the next subsection.

C_1	C_2	C_3	C_4	C_5	C_6
0	0	2	2	0	0
1	1	3	3	8	8
1	2	2	2	4	4
2	3	1	1	9	9
2	4	2	2	2	2
2	5	3	3	10	10
2	6	2	2	5	5
2	7	4	4	11	11
2	8	2	2	1	1
2	9	3	3	12	12
2	10	2	2	6	6
3	11	1	1	13	13
3	12	2	2	3	3
3	13	3	3	14	14
3	14	2	2	7	7
4	15	0	0	15	15

Figure 2: R_4 relation

4.2.2 Properties of the Database

Before we describe the data, let us present some definitions. Let $SEQ(n,i)$ (similarly $SIQ(n,i)$) be a select query on R_n using an equality (similarly inequality) predicate with the cardinality of the output result being 2^i for $i < n$. Such a query will be of use in the following discussion.

The values in C_1 are in ascending order with a clustered index. The $f[i]$ gives the number of tuples in which the value i occurs in $C1$. Let us define $mf[i]$ to be the ith most frequently occurring value in C_1. The distribution of values exhibits a 'normal' pattern such that the $mf[i]$ value occurs in $1/2^i$ of the number of tuples. The formulae for $C_1[n,i]$ above has encoded this pattern, from which we can make the following lemma.

Lemma 1 *For any relation R_n, and any selectivity $s_i = 1/2^i, i \in [1,n]$, there exists an equality predicate on C_1 whose selectivity is s_i.*

Corollary 1 *There exists queries $SEQ(n,i)$ and $SEQ(n+1,i)$ on relations R_n and R_{n+1} respectively for $i = 1, 2, \ldots, n$.*

The above observations provide us with the guarantee that queries exist that

- select varying number of tuples using equality predicate on the same relation; and
- select the same number of tuples using equality predicate from multiple relations of different sizes.

For these queries using C_1, the following claim can be argued with reasonable assurance.

Claim 1 *Execution of any $SEQ(*,*)$ query on C_1 will result in the use of the clustered index scan.*

As mentioned before, the predictability of the cost of executing a query $SEQ(n,i)$ on C_1 will depend on the CPU and I/O components[4]. CPU cost increases monotonically with the size of the result. Because of the above claim, the I/O cost also increases monotonically until the maximum number of pages are accessed. Thus the number of I/O is a nondecreasing function of the size of the result. We shall refer to this class of function as *saturating monotonically increasing function.*

Lemma 2 *For any storage implementation, and page size used, the number of pages accessed by a SEQ(n,i) query using C_1 will be given by a saturating monotonically increasing function on i.*

Note that calibration of the cost formulae in the saturated region will result in incorrect calibration. However, observe that almost half the number of pages will not be accessed in the worst case. Therefore, if we use large relations, the problem of saturation can be avoided. Thus, we can conclude that unpredictability problems are avoided for any query on C_1.

Note that the values in C_1 are functionally determined by the value in C_2[5]. In fact the values of C_2 are in ascending order because of the ascending-order indexing of the key. This observation and the fact that the index is clustered lead us to the following lemma.

Lemma 3 *Any storage implementation of this relation will retain this order of the tuples amongst pages of the relation.*

This observation gives us a handle on the pagination of the data. We shall use this to argue that C_3 and C_4 values are uniformly distributed across all pages.

The following two observations are obvious but stated here for completeness.

[4]We omit communication cost for simplicity. This can be also included without loss of generality.

[5]For this reason, this relation is not in 2NF. As we are not concerned with updates to this relation, lack of normalization is not of concern.

Lemma 4 *For any relation R_n, and any selectivity $s_i = 1/2^i, i \in [1,n]$ there exists an inequality predicate on C_2 whose selectivity is s_i.*

Corollary 2 *There exists queries $SIQ(n,i)$ and $SIQ(n+1,i)$ on relations R_n and R_{n+1} respectively for $i = 1,2,\ldots,n$.*

Let us observe that the values in C_1 are permuted into different rows of C_3. Therefore, the frequency distribution $f[i]$ also applies to C_3 and C_4. This redistribution is done with the observation that $\frac{1}{2}$ of the number of tuples have row index in binary representation the pattern *0 (i.e., last bit is zero), $\frac{1}{4}$ of the number of tuples have row index in binary representation the pattern *01, etc. Therefore, the distribution can be done as follows:

- All row index in binary that has the pattern *0 has $mf[1]$ value.
- All row index in binary that has the pattern *01 has $mf[2]$ value.
- All row index in binary that has the pattern *011 has $mf[3]$ value.
- *...etc....*

Thus we can conclude that any value in $[0,n]$ is uniformly distributed in the rows for C_3 and C_4. This leads us to the following lemma.

Lemma 5 *For any storage implementation, and page size used, and given a value $i \in [0,n]$, tuples containing value i for C_3 and C_4 are uniformly distributed amongst all the pages.*

Using this lemma we can make the following observation that overcomes one predicatability problem when using C_3 or C_4.

Lemma 6 *For any storage implementation, and page size used, the number of pages accessed by a SEQ(n,i) query using C_3 or C_4 will be given by a saturating monotonically increasing function on i.*

Once again we argue that saturating is not a problem because the unclustered index is mostly useful in the region when the selectivity is low. So if the calibration is restricted to this region then the I/O will be monotonically increasing. With CPU cost increasing monotonically, we have avoided one of the predicatability problem.

In order to determine the region when the index is being used we make the following observations.

Claim 2 *Execution of any $SEQ(*,*)$ query on C_3 will result in the use of the index scan if the selectivity is low; but if the selectivity is high then the system may use sequential scan.*

Claim 3 *Execution of any $SEQ(*,*)$ query on C_4 will result in the use of the sequential scan.*

Knowing that C_3 and C_4 are identical, we can easily determine the region of selectivities when index is being used and use that data to calibrate the system. Thus we can conclude that unpredictability problems are avoided for queries on C_3 and C_4.

C_5 and C_6 are also permutations of C_2 and are intended for use with inequality queries. So the predicatability problem for such a query involves the number of pages accessed by a selection with an inequality predicate of the form $C_5 < i$. The values need to be distributed in such a fashion that the number of pages accessed are increasing as the selectivity is increased. This is achieved by distributing the values with the following property.

Lemma 7 *For any $i \in [0, n]$, the set of values $[0, 2^i - 1]$ are distributed uniformly in C_5 and C_6.*

Intuitively, this results in requiring a $SIQ(n,i)$ query to access a sequence of row indices such that successive row indices differ by a constant. Consider a query $C_5 < 8$. This will access the rows (0, 2, 4, 6, 8, 10, 12, and 14) in R_4 with the property that successive rows differ by 2. Note that the condition is of the form $C_5 < 2^i$. Using this observation we can state the following lemma.

Lemma 8 *For any storage implementation, and page size used, the number of pages accessed by a $SIQ(n,i)$ query using C_5 or C_6 will be given by a saturating monotonically increasing function on i.*

Once again using the argument similar to the one used for C_3 and C_4 we can argue that predicatability problems can be avoided. The above observation for the relation R_n is particularly important because such a conclusion cannot be made if the relation is generated probabilistically as it was done in the benchmarking studies.

In summary we have argued that the queries posed against any of the attributes in the relation have predictable behavior.

4.2.3 Calibrating Procedure

Now we proceed to show the procedure to deduce the coefficients from the observed execution of these queries.

Claim 4 *Cost of execution of queries $SEQ(n,i)$ and $SEQ(n+1,i)$ are identical except for the $COMP_1$ component of the cost due to the fact that they are accessing relations of different sizes.*

From this observation we can construct the following experiment:

- Evaluate $SEQ(n,i)$ and $SEQ(n+1,i)$ using an equality predicate on C_4 and observe the cost.
- Knowing that the system will choose sequential scan, solve for the coefficient $CS1_{ss}$.

Note that the above experiment is not to be done with one or two points. Rather, many values are to be used to get the value for the coefficient so that error is minimized. These observations will be elaborated in the experimentation section.

Claim 5 *Cost of execution of queries $SEQ(n,i)$ and $SEQ(n,i+1)$ are identical except for the $COMP_2$ component of the cost due to the fact that they are selecting different number of tuples.*

From this observation we can construct the following experiment:

- Evaluate $SEQ(n,i)$ and $SEQ(n,i+1)$ using an equality predicate on C_4 and observe the cost.
- Knowing that the system will choose sequential scan, solve for the coefficient $CS2_{ss}$.

Similar experiment on C_1 and C_3 can compute the coefficient $CS2_{ci}$ and $CS2_{ui}$ for clustered index and unclustered index respectively. As before, by projecting only the value for C_1 with selection for C_1, the coefficient $CS2_{is}$ can be computed using a similar procedure as above.

Knowing $CS1_{ss}$ and $CS2_{ss}$ for the sequential scan cost formula, we can compute $CS0_{ss}$ from the respective of the observed cost for SEQ(*,*).

The bifurcation of $CS1_{ss}$ into $CS1^{io}_{ss}$ and $CS1^{cpu}_{ss}$ is done by scanning a table twice in the same query in the manner specified by the following SQL query:

```
select t1.C6 from Rn t1,Rn t2
where t1.C6=t2.C6 & t1.C6<c
```

Note that computing $CS1_{is}, CS1_{ci}$ and $CS1_{ui}$ poses a problem. Because these values are expected to be small compared to other components and factoring them to a reasonable degree of accuracy is difficult. So we make the assumption that these coefficients have the same value as $CS2_{ci}$. This is because the main cost of initial index tree lookup is the amortized I/O cost of fecthing index page. Since the index tree is usually clustered, this cost should be similar to that of fetching data page for clustered index, i.e., $CS2_{ci}$. Our validation on Allbase, DB2, and Informix also corroborated this point of view.

Thus we can compute the coefficients in the cost formulae for selection using a series of observations from queries.

Further note that similar experiments can be done using C_2, C_5 and C_6 to compute the coefficients for the inequality select operation.

Next we outline the method to determine the cost of ordering a relation. This is needed in the ordered merge cost formulae. This is done by joining two R_n's, in the manner specified by the following SQL query:

```
select t₁.C₄, t₂.C₄ from R¹ₙ t₁ R²ₙ t₂
where t₁.C₄ = t₂.C₄ & t₁.C₆+t₂.C₆<c
```

Note that the first condition is preferable for as the join condition and C_4 does not have an index. Therefore, ordered merge algorithm will be used to compute the join. The output can be varied with the appropriate choice of the constant. Using the observation for queries on one relation with varying size of the output, the constant $CJ2_{mg}$ can be deduced. Further the same query can be computed for three or four values of n and the cost of sorting the relation can be deduced from the knowledge of the cost of sequential scan.

Using the cost formulae for selection and ordering, we can compute the cost of joins without predicatability problems.

Theorem: *The coefficients of the cost formulae are computed without the unpredictability problems.*

Note that the viability of this approach is predicated on the following two assumptions.

- Some relations can be stored in the participating DBMS, either as a multidatabase user of the system or if such privilege is not available to the multidatabase user then by special request to the database administrator. These relations are to be used temporarily for calibration and not needed after calibration.
- The observed behavior of the queries are repeatable in the sense that the effect of other concurrent processes do not distort the observations.

Relation Name	Relation Type	Size of tuple	Card. of Relation
t1	R_{10}	42	2^{10}
t2	R_{10}	42	2^{10}
t3	R_{10}	84	2^{10}
t4	R_{13}	42	2^{13}
t5	R_{13}	42	2^{13}
t6	R_{13}	84	2^{13}
t7	R_{15}	42	2^{15}
t8	R_{15}	42	2^{15}
t9	R_{15}	84	2^{15}
t10	R_{17}	42	2^{17}
t11	R_{17}	42	2^{17}
t12	R_{17}	84	2^{17}
t13	R_{20}	42	2^{20}
t14	R_{20}	42	2^{20}
t15	R_{20}	84	2^{20}

Figure 3: Calibrating Relations

4.3 Practical Calibrations

In the last section we outlined a procedure to calculate the coefficients of the cost formulae. We use this technique to caliberate three commercial DBMSs — Allbase, DB2, and Informix; i.e., compute the coeffiecients of the cost formulae. The experiments are set up so as to use mostly single table queries. This is not only because the join queries are time consuming and therefore takes too long to caliberate the system, but also because the cost of most join queries can be estimated using those of single table queries. As a validation, we ran various kinds of join queries and compared the estimated cost with the actual observed cost. The result showed that the coefficients can be estimated to the extent that subsequent estimation of the join queries were within 20% of actual observations. In this subsection we discuss the queries posed and the calibrated coefficients. Finally we discuss the verification queries and the results.

The systems calibrated were an Allbase DBMS (version HP36217-02A.07.00.17) running on an HP 835 Risc workstation, a DB2 DBMS (version V2.2) running on an IBM 3090 Mainframe and an Informix DBMS (version 4.00.UE2) running on an HP 850 Risc workstation. The calibrations were done at night when DB2 was comparably lightly loaded whereas the Allbase and Informix had no other contender. DB2 and Informix DBMSs were intended for production use and were set up to suit their applications. For that reason as well as to respect the autonomy of that installation,

1.1. select C_1 from R_n where $C_1 = c$
1.2. select C_3 from R_n where $C_3 = c$
1.3. select C_4 from R_n where $C_4 = c$
2.1. select C_2 from R_n where $C_1 = c$
2.2. select C_2 from R_n where $C_3 = c$
2.3. select C_2 from R_n where $C_4 = c$
3.1. select C_2 from R_n where $C_2 < c$
3.2. select C_5 from R_n where $C_5 < c$
3.3. select C_6 from R_n where $C_6 < c$
4.1. select C_1 from R_n where $C_2 < c$
4.2. select C_1 from R_n where $C_5 < c$
4.3. select C_1 from R_n where $C_6 < c$

Figure 4: Calibration Queries for Table R_n

Const.	Allbase	DB2	Informix
$CS0_{ss}$	1.9	0.60	0.06
$CS1_{ss}$	ts/10^5	ts/1.4*10^6	(168+ts)/7*10^5
$CS2_{ss}$	0.000175	0.0003	0.00045
$CS0_{is}$	1.35	1.2	0.06
$CS1_{is}$	0.0007	0.0003	0.001
$CS2_{is}$	0.00036	0.0003	0.001
$CS0_{ci}$	1.35	1.2	0.06
$CS1_{ci}$	0.0007	0.0003	0.001
$CS2_{ci}$	0.0007	0.0003	0.001
$CS0_{ui}$	1.35	1.2	0.06
$CS1_{ui}$	0.0007	0.0003	0.001
$CS2_{ui}$	.014 - .024	.007 - .009	.001 - .002

Note:
- ts is the tuple size (in bytes).
- $CS2_{ui}$ varies slightly on table size.

Figure 5: Allbase, DB2, and Informix Cost Formulae

the system parameters were not altered.

There are 16 relations used in these calibrations as shown in Figure 3. Each type of relation was instantiated with two sizes of tuples and the smaller tuple relation was duplicated. This duplication is because the join queries needed two identical relations. Relations of type R_{10}, R_{13}, R_{15} *and* R_{17} were used in the calibration of Allbase and Informix whereas relations of type R_{13}, R_{15}, R_{17} *and* R_{20} were used in the calibration of DB2. This choice is because of the availability of disk space. The calibration procedure is identical.

The actual queries used in the calibration are given in Figure 4, where R_n is a table of cardinality 2^n and c is a constant which determines the selectivity. For each type of query against R_n, a set of queries with selectivity 2^{-i} (i=1,2,...,n) are constructed and observed.

For each query the elapsed time in the DBMS is recorded. For DB2, the elapsed time is defined as class 2 elapse time (see [IBM-DB2]). For Allbase and Informix, it is calculated by subtracting the start timestamp (when the query is received) from the end timestamp (when the result has been sent out). In all DBMSs the queries were posed after flushing all the buffers to eliminate the distortion due to buffering. Each query is issued 30 times and the average elapse time is calculated. Except few cases, relative error between actual data and average value is within 5% with confidence coefficient of 95%. Thus, the repeatability of the observation was assured.

From these, the coefficients for the cost formulae for Allbase, DB2, and Informix can be deduced. Least square fitting algorithm was used to minimize the errors and estimate the coefficients. These are reported in Figure 5. The elapsed time for queries of the type 3.1 running on DB2 for relations R_{13} and R_{17} along with the estimated time by the cost formula, which is independent of the size of the relation, are shown in Figure 7. This corroborates two facts: 1) DB2 access to index only queries are not sensitive to size of the relation; 2) the approximation used for $CS1_{is}$ is not affecting the accuracy very much. As it can be seen, the error is quite small and this was true for all tests using the queries above.

To explore the effect of multiple selection and projection clauses, we modify and rerun the twelve basic queries on DB2. The new queries have upto five predicates and return upto five attributes. The following are two example queries.

```
select C1,C2,C3,C4,C5 from Rn where C5<c
```

```
Select C5 from Rn
where C5<c & C1≥0 & C2≥0 & C3≥0 & C4≥0
```

Note that in the second query, the '≥' predicates are true for all tuples. This guarantees that they are checked for those and only those tuples that have succeeded with the first (original) predicate.

The result of the experiments shows that in all cases, the differences are within 10%. It seems to suggest that the cost of projecting additional columns is negligible once the tuple is in the memory and the cost to check predicates is also minimal comparing to other processing costs (e.g., I/O).

The above cost formulae were also validated using the 36 types of join queries shown in Figure 6. Queries of type 1.1-2.9 return 2^{2*i} tuples depending on constant c, where $i = 0, 1, \ldots, n$. In this validation, joins of selectivity 2^{2*k} for $k < (n-4)$ were tested. Queries of type 3.1-4.9 return 2^i tuples depending on c, where $i \leq$

1.1. select $R_n.C_1, R_m.C_1$ where $R_n.C_1 = c$ & $R_n.C_1 = R_m.C_1$
1.2. select $R_n.C_3, R_m.C_3$ where $R_n.C_3 = c$ & $R_n.C_3 = R_m.C_3$
1.3. select $R_n.C_4, R_m.C_4$ where $R_n.C_4 = c$ & $R_n.C_4 = R_m.C_4$
1.4. select $R_n.C_4, R_m.C_1$ where $R_n.C_4 = c$ & $R_n.C_4 = R_m.C_1$
1.5. select $R_n.C_4, R_m.C_3$ where $R_n.C_4 = c$ & $R_n.C_4 = R_m.C_3$
1.6. select $R_n.C_1, R_m.C_4$ where $R_n.C_1 = c$ & $R_n.C_1 = R_m.C_4$
1.7. select $R_n.C_1, R_m.C_3$ where $R_n.C_1 = c$ & $R_n.C_1 = R_m.C_3$
1.8. select $R_n.C_3, R_m.C_1$ where $R_n.C_3 = c$ & $R_n.C_3 = R_m.C_1$
1.9. select $R_n.C_3, R_m.C_4$ where $R_n.C_3 = c$ & $R_n.C_3 = R_m.C_4$
2.1. select $R_n.C_2, R_m.C_2$ where $R_n.C_1 = c$ & $R_n.C_1 = R_m.C_1$
2.2. select $R_n.C_2, R_m.C_2$ where $R_n.C_3 = c$ & $R_n.C_3 = R_m.C_3$
2.3. select $R_n.C_2, R_m.C_2$ where $R_n.C_4 = c$ & $R_n.C_4 = R_m.C_4$
2.4. select $R_n.C_2, R_m.C_2$ where $R_n.C_4 = c$ & $R_n.C_4 = R_m.C_1$
2.5. select $R_n.C_2, R_m.C_2$ where $R_n.C_4 = c$ & $R_n.C_4 = R_m.C_3$
2.6. select $R_n.C_2, R_m.C_2$ where $R_n.C_1 = c$ & $R_n.C_1 = R_m.C_4$
2.7. select $R_n.C_2, R_m.C_2$ where $R_n.C_1 = c$ & $R_n.C_1 = R_m.C_3$
2.8. select $R_n.C_2, R_m.C_2$ where $R_n.C_3 = c$ & $R_n.C_3 = R_m.C_1$
2.9. select $R_n.C_2, R_m.C_2$ where $R_n.C_3 = c$ & $R_n.C_3 = R_m.C_4$
3.1. select $R_n.C_2, R_m.C_2$ where $R_n.C_2 < c$ & $R_n.C_2 = R_m.C_2$
3.2. select $R_n.C_5, R_m.C_5$ where $R_n.C_5 < c$ & $R_n.C_5 = R_m.C_5$
3.3. select $R_n.C_6, R_m.C_6$ where $R_n.C_6 < c$ & $R_n.C_6 = R_m.C_6$
3.4. select $R_n.C_6, R_m.C_2$ where $R_n.C_6 < c$ & $R_n.C_6 = R_m.C_2$
3.5. select $R_n.C_6, R_m.C_5$ where $R_n.C_6 < c$ & $R_n.C_6 = R_m.C_5$
3.6. select $R_n.C_2, R_m.C_6$ where $R_n.C_2 < c$ & $R_n.C_2 = R_m.C_6$
3.7. select $R_n.C_2, R_m.C_5$ where $R_n.C_2 < c$ & $R_n.C_2 = R_m.C_5$
3.8. select $R_n.C_5, R_m.C_2$ where $R_n.C_5 < c$ & $R_n.C_5 = R_m.C_2$
3.9. select $R_n.C_5, R_m.C_6$ where $R_n.C_5 < c$ & $R_n.C_5 = R_m.C_6$
4.1. select $R_n.C_1, R_m.C_1$ where $R_n.C_2 < c$ & $R_n.C_2 = R_m.C_2$
4.2. select $R_n.C_1, R_m.C_1$ where $R_n.C_5 < c$ & $R_n.C_5 = R_m.C_5$
4.3. select $R_n.C_1, R_m.C_1$ where $R_n.C_6 < c$ & $R_n.C_6 = R_m.C_6$
4.4. select $R_n.C_1, R_m.C_1$ where $R_n.C_6 < c$ & $R_n.C_6 = R_m.C_2$
4.5. select $R_n.C_1, R_m.C_1$ where $R_n.C_6 < c$ & $R_n.C_6 = R_m.C_5$
4.6. select $R_n.C_1, R_m.C_1$ where $R_n.C_2 < c$ & $R_n.C_2 = R_m.C_5$
4.7. select $R_n.C_1, R_m.C_1$ where $R_n.C_2 < c$ & $R_n.C_2 = R_m.C_6$
4.8. select $R_n.C_1, R_m.C_1$ where $R_n.C_5 < c$ & $R_n.C_5 = R_m.C_2$
4.9. select $R_n.C_1, R_m.C_1$ where $R_n.C_5 < c$ & $R_n.C_5 = R_m.C_6$

Figure 6: Join Queries for Tables R_n and R_m

$min(n, m)$. Joins of selectivity 2^k for $k < min(n, m) - 4$ were tested. In both cases, the observed value was compared with the estimated value.

In Figure 8, we show the comparison of the estimated value with the observed values for the type 3.1 join queries running on DB2 using pairs of relations $R^1_{13} \bowtie R^2_{13}$, $R^1_{17} \bowtie R^2_{17}$ and $R_{13} \bowtie R_{17}$. Note that, in this case, the DB2 chooses nested loop as the join method and index-only access method. Therefore, in all the three cases the cost should be independent of the cardinality of the two relations. This is observed from the graph shown in Figure 8 wherein the maximum error is about 10%. Once again corroborating the cost model and approximations.

Figure 9 and 10 show the joins using sort-merge with $R^1_{17} \bowtie R^2_{17}$ and $R_{13} \bowtie R_{17}$ on DB2 respectively. Note that the estimated cost is once again within 10% error.

The results of this validation showed that in more than 80% of the cases the observed value was within a band of 20% error from the estimated value. Further, in all the other cases the following phenomenon contributed to the majority of the error. All these cases were when the system used unclustered index in the inner loop of the nested loop join. As the cost CS_{ui} is computed as a stand alone query, the potential buffering of pages underestimated the cost of this access in the inner loop of the join where it is competing for buffers with the outer loop. Thus, the estimate was always lower. We believe this can also be corrected and is a topic of future research.

5 Conclusion

In this paper we have proposed to use the traditional architecture for query optimization wherein a large execution space is searched using dynamic programming strategy for the least cost execution based on a cost model. It is shown that the traditional execution space can be simply extended to allow for remote joins and the search strategy can be used as is in the new context of heterogeneous DBMSs. Thus the crux of the problem is to design cost models for different DBMSs such that they can be used by the heterogeneous query optimizer.

The design of cost model for proprietary DBMSs is an important problem but does not pose any new challenges in our opinion. In contrast deriving the cost model for conforming and non-conforming DBMSs while treating the DBMS as a black box is the challenge. In this paper, we proposed a calibration procedure for conforming DBMSs. A calibrating database is created in the DBMSs that are to be calibrated and has properties that enable the system to make observations while avoiding unpredictability problems resulting from the black box nature of interaction.

This calibrating procedure has been used to calibrate three commercial DBMSs (Allbase, DB2, and Informix) and the results are promising. The estimated values for majority of the queries were within 20% error of the observed values. The remaining ones with more error were of a particular type that used unclustered index in nested loop joins. Being the first attempt to calibrate a database system using an arm-length procedure, this is not only novel but also promising.

This work is definitely a beginning of more research that needs to be done for this approach to be successful. First we would like to calibrate a few more DBMSs such as Oracle and Sybase. As mentioned before, it is possible to improve the cost model by incorporating more aspects of the underlying system.

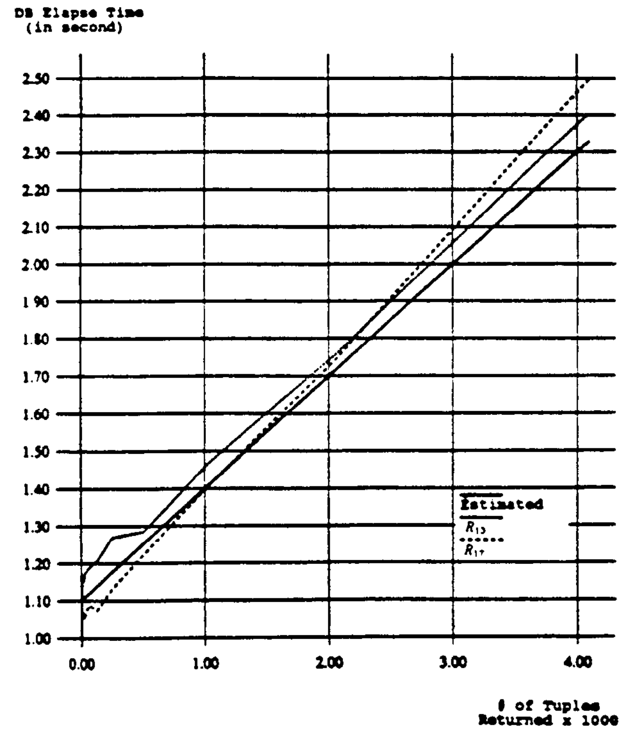

Figure 7. select * from R_m where $R_m.C_3 < c$

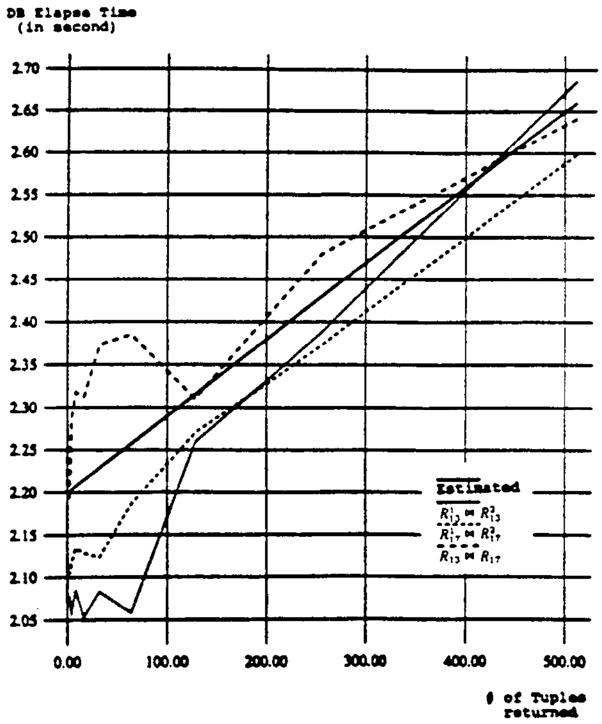

Figure 8. select * from R_m, R_m where $R_m.C_2 < c$ and $R_m.C_3 = R_m.C_3$

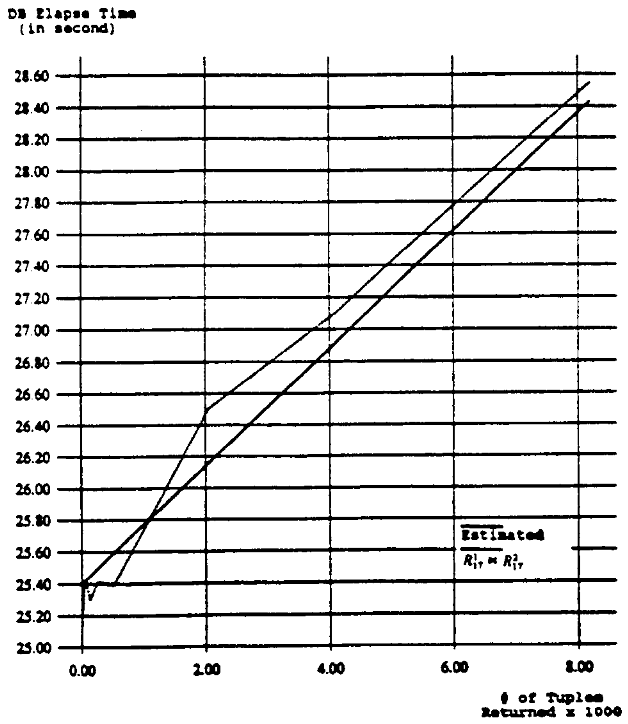

Figure 9. select * from R^1_{17}, R^2_{17} where $R^1_{17}.C_6 < c$ and $R^1_{17}.C_6 = R^2_{17}.C_6$

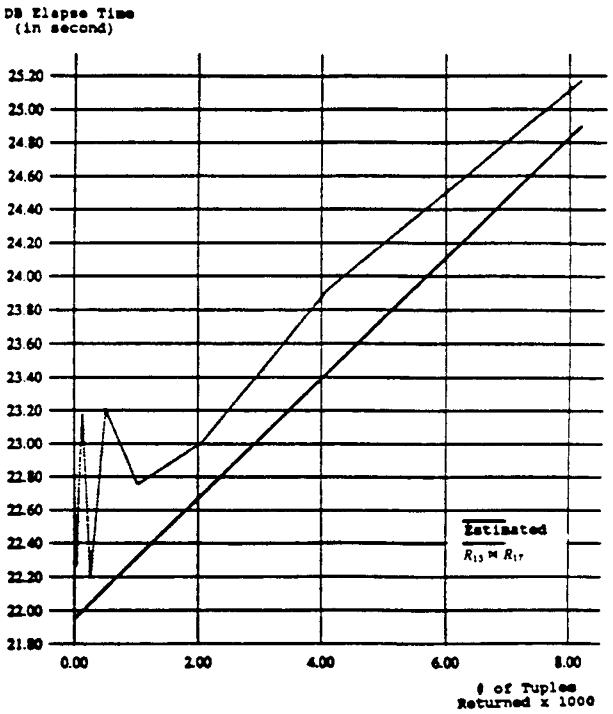

Figure 10. select * from R_{13}, R_{17} where $R_{13}.C_6 < c$ and $R_{13}.C_6 = R_{17}.C_6$

- Quantifying the effect of buffering on unclustered index is an important extension. As mentioned before, almost all the queries that had significant error could be argued to be due to this problem.
- Inferring de facto clustering from the data, such as C_2 in the database would be quite useful. Systems such as DB2 have knowledge of the degree of clustering for this column to be 100% and therefore may use it. In general, such information can still be used to interpolate the cost of 'less clustered' data.
- Detecting when an index scan switches to sequential scan for an unclustered index is an important information if this access method is used in a nested loop. As the system may be using some rule of thumb for the switch, the HDBMS optimizer should know the point of transition.

Finally the approach proposed here was in the context of a particular cost model. It should be obvious to the reader that even if the cost model is changed the approach can still be used to devise a new procedure. Thus, we feel the process of designing a synthetic database in a deterministic manner has merit.

Using the knowledge of the proposal here for conforming DBMSs, it is possible to extrapolate an approach for the non-conforming DBMSs. We are developing a single invocation and execution model for DBMSs in all three categories so that the query optimization process developed in this paper can be extended to cover non-conforming DBMSs. To serve the purpose, we plan to integrate some widely used non-conforming DBMSs such as IMS, VSAM and Unix applications. The calibration procedure may need to be enhanced to measure DBMSs of this category.

We also plan to explore issues of post query optimization such as dynamic reconfiguration of execution plan at run time. This is important, especially in HDBMSs, because it is difficult (and sometimes impossible) to get accurate cost formulae. Other related issues include reduction of invocations to participating DBMSs, cross site data buffering, new global join methods, etc. These are topics of future research.

Acknowledgements

We wish to acknowledge the following people in HP Corporate Computing and Services for their help in our calibration. Mike Wang and Arthur Walasek of SOLID project provided accesses to DB2 and Informix. Charlene Yoshizu, Jeff Litalien, and Kim Iverson provided considerable system support and patience while our calibration consumed vast amounts of their resources.

References

[Aetal91] Ahmed, R., et al, "The Pegasus Heterogeneous Multidatabase System", IEEE Computer Magazine, Vol. 24, No. 12, 1991.

[BDT83] Bitton, D., D. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic approach" Proc. of VLDB, 1983.

[CGK90] Chimenti, D., R. Gamboa, and R. Krishnamurthy, "Abstract Machine for $\mathcal{LDL}$," Proc. of EDBT, 1990.

[CP84] Ceri, S. and G. Pelagatti, *Distributed Databases: Principles & Systems*, McGraw-Hill Book Company, 1984.

[Da85] Dayal, U., "Query Processing in Multidatabase System", *Query Processing in Database Systems*, Edited by W. Kim, D. Reiner and D. Batory, Springer Verlag, 1985.

[GHK92] Ganguly, S., W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution", Proc. of SIGMOD, 1992.

[HWKSP1] Proc. of Workshop on Multidatabases and Semantic Interoperability, Tulsa, OK., Nov. 1990, Tulsa, OK.

[HWKSP2] Proc. of 1st Intl. Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan, Apr., 1991.

[IBM-DB2] "DB2 Performance Monitor (DB2PM) Usage Guide", Document no. GG24-3413-00, Published by IBM Corporation, 1989.

[KBZ86] Krishnamurthy, R., H. Boral, and C. Zaniolo, "Optimization of Non-Recursive Queries," Proc. of VLDB, 1986.

[LS92] Lu, H., and M. Shan, "Global Query Optimization in Multidatabase Systems", 1992 NFS Workshop on Heterogeneous Databases and Semantic Interoperability, 1992.

[Sell79] Sellinger, P.G. et al., "Access Path Selection in a Relational Database Management System," Proc. of SIGMOD, 1979.

[WK90] Whang, K., and R. Krishnamurthy, "Query Optimization in a Memory-Resident Domain Relational Calculus Database System" TODS, Vol, 15, No. 1, 1990.