A Formal Model of Trade-off between Optimization and Execution Costs in Semantic Query Optimization

Shashi Shekhar
Jaideep Srivastava
Soumitra Dutta

Computer Science Division,
University of California,
Berkeley, CA 94720.

ABSTRACT

Conventional query optimizers assume that the cost of optimization is negligible. This assumption does not hold for much larger search spaces (of possible execution plans) such as those encountered during semantic query optimization. In particular, the optimization cost can become comparable to the execution cost, and thus a significant fraction of the response time for interactive queries[1]. This paper discusses the tradeoff between the two costs in the context of semantic query optimization, and reports a heuristic search algorithm which minimizes a weighted sum of both the costs. A detailed analysis of an experiment is presented to strengthen the claim. The paper also contributes a practical model of semantic query optimization, and a discussion of its search ordering and termination problems.

1. Introduction

Conventional query optimization is based on syntactic rearrangements [2], query decomposition [3], and optimal usage of indices, join algorithms and database statistics [4]. Several query execution plans are examined to select the minimum cost plan. These methods are not flexible enough to generalize to new applications, extensible databases, and also there is no mechanism to use application specific knowledge for optimization. For example, user-defined data-types make it difficult for an optimizer to reason about the *value* range of data-items in a relation. Also current optimizers are unable to handle user defined types, operators and access methods, as it is difficult to estimate the cost of computing *operators* in different ways, since optimizer doesn't know the details of the access methods. Similarly, it is difficult to optimize *procedure* valued fields in a tuple using conventional techniques.

Some of these difficulties can be alleviated by *Semantic Query Optimization (SQO).* SQO uses semantic information about the database, *eg.* semantic integrity constraints and functional dependencies, for optimization. The original query is transformed into syntactically different, but *semantically equivalent* † queries, which may possibly yield a more efficient execution plan[5]. Semantic query optimization also provides the flexibility to add new information and optimization methods to an existing optimizer. A modular arrangement of optimization methods makes it possible to add, delete and modify individual methods, without affecting the rest. This provides an extensible system for maintaining and managing optimization strategies, as it is implemented as a rule-based system. Semantic query optimization is well motivated in the literature[6,5,7], as a new dimension to conventional query optimization.

However, semantic optimization increases the search space of possible plans by an order of magnitude, and very efficient searching techniques are needed to keep the cost of optimization within reasonable limits. Moreover, as the semantic information about the database (and thus the corresponding space of semantically equivalent queries) increases, the optimization cost becomes comparable to the cost of query execution plan, and cannot be ignored. The tradeoff between optimization time and the quality of query execution plans, becomes a major issue in minimizing the total cost of query processing.

There has been little research in controlling the searching costs, and trading optimization time with the quality of the execution plan. Simple schemes to locate relevant integrity constraints have been proposed[8], but those neither use good algorithms for searching in the space of possible plans, nor consider the tradeoff between optimization and execution costs. As far as we know this paper is the first attempt towards *(i)* presenting efficient heuristic algorithms for reduce searching during semantic query optimization, and *(ii)* formalizing the trade-off between the optimization cost incurred and the quality of execution plan obtained.

The total cost of *query evaluation* has two parts: (a) *optimization cost* to select the query execution plan, and (b)

† Semantically equivalent queries produce the same answer for all database instances that satisfy the integrity constraints and functional dependencies.

execution cost to run the execution plan. Conventional query optimizers assume that the first part is negligible compared to the second, and they try to minimize only the execution cost instead of the total query evaluation cost. Ignoring optimization cost is no longer reasonable if the space of all possible execution plans is very large as those encountered in SQO[8] as well as in optimization of queries with a large number of joins. The optimization cost becomes comparable to query execution cost, and minimizing execution cost alone would not minimize the total cost of query evaluation, as illustrated in *Fig 1.1.*

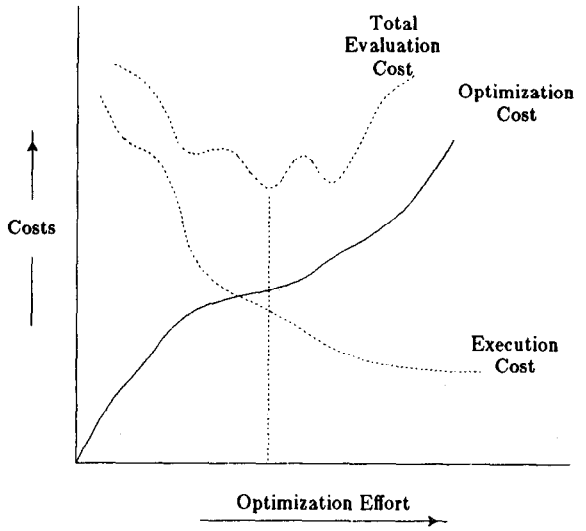

Fig. 1.1 *Trading optimization cost with execution cost*

Outline of the paper: The problem is formally defined in section 2, and the notation for subsequent discussion is introduced. Section 3 presents a brief survey of heuristic searching techniques from Artificial Intelligence. Section 4 describes the issues of generating semantically equivalent queries, selecting the best candidate for exploration, and search termination criteria. The near-optimal searching algorithm for trading the optimization cost with the execution cost, is presented in subsection 4.3. In section 5 we present the results of an experiment to validate the algorithm.

2. Problem Definition

This section summarizes the notation used in this paper and provides a formal definition of the problem. The notation is summarized in *Table 2.1.*

We present below formal definitions of Conventional Query Optimization and Semantic Query Optimization, both to differentiate between the two precisely and to make the discussion of the latter more concrete.

Conventional Query Optimization (CQO): This is the problem of finding the minimum cost query execution plan from the set of all possible plans for the query *as posed by the user.*

Given: Query Q_0

Required:

Find the least cost plan, $qp(0,b)$, for query Q_0 such that $C_E(0,b) \leq C_E(0,j)$ for $j \geq 1$.

Symbol	*Meaning*
Q_0	*query posed by user*
$Q_i,\ i \geq 1$	*various queries semantically equivalent to* Q_0
$QP(i),\ i \geq 0$	*set of Query Plans for* Q_i
$qp(i,j),\ i \geq 0;\ j \geq 1$	*elements of* $QP(i)$
$C_E(i,j),\ i \geq 0;\ j \geq 1$	*estimated execution cost of* $qp(i,j)$
$qp(i),\ i \geq 0$	*least cost element of* $QP(i)$
$C_E(i)$	*estimated execution cost of* $qp(i)$ *(provided by conventional query optimizers)*†
$SP(i)$	*space of queries already explored upto and including* Q_i
$C_E^{\min}$	*least estimated execution cost so far,* $C_E^{\min} = \min_{j \in SP(i)} C_E(j)$
λ	*factor of relative importance between total optimization cost and least execution cost*
$C_O^r(i)$	*cost of applying some rule to reach* Q_i *from immediately preceding semantically equivalent query*
$C_O^s(i)$	*cost of running the conventional optimizer on* Q_i *to find least cost evaluation plan for it*
$C_O(i)$	$C_O^r(i) + C_O^s(i)$

Table 2.1. *Summary of Notation Used.*

Semantic Query Optimization (SQO): This is the problem of finding the minimum cost query execution plan from the set of all possible plans for *all the queries that are semantically equivalent ‡ to the user's query.*

Given: Query Q_0

Required:

Find least cost plan, $qp(a)$, for query Q_a such that $C_E(a) \leq C_E(i)$ for all Q_i's.

Integrated Semantic Query Optimization (ISQO): This is the problem of searching the space of all possible query execution plans for all the semantically equivalent queries, but stopping the search when the total query evaluation time (i.e. optimization cost so far + execution cost) is minimum.

Given: Query Q_0

Required:

Find plan $qp(a)$ for query Q_a such that

$$\sum_{Q_j \in SP(a)} C_O(j) + \alpha \overline{\overline{C_E}}(a) \leq \sum_{Q_j \in SP(i)} C_O(j) + \alpha C_E(i) \quad \text{for all } Q_i\text{'s.}$$

† Conventional query optimizers like R^* do not perform any kind of semantic query optimization.

‡ A set of queries is defined to be *semantically equivalent* w.r.t. to a set of logical constraints. The set of constraints we are considering are the semantic integrity constraints of the database.

‡‡ Query evaluation cost is a weighted sum of the optimization cost and the execution cost, i.e. $C_O + \alpha C_E$. When α==1, query evaluation cost is the same as the query response time (ignoring queuing delays).

The problem, as stated above, is extremely difficult. Thus, the approach we take is to reduce the complexity of the problem by settling for a near-optimal solution instead of the optimal one. This can be done by limiting the search to *most* Q_i's instead of *all* Q_i's.

3. The Search Space & Searching Techniques

This section first describes the nature of the space of query execution plans that the semantic query optimizer has to search. It then discusses various search algorithms.

3.1. The Search Space

Conventional query optimizers select the optimal query execution plan by searching through $QP(0)$ only, the space of query plans corresponding to the original user query, Q_0. In contrast, semantic query optimizers search the much larger space of execution plans, QP_{tot}, where,

$$QP_{tot} \equiv QP(0) \bigcup QP(1) \bigcup \cdots \bigcup QP(n)$$

One conceptual way to model the search space is in terms of the following two levels:

[1] *Level 1:* The space of semantically equivalent but syntactically different queries, $Q_0, Q_1, \cdots Q_n$.

[2] *Level 2:* The spaces $QP(0), QP(1), ..., QP(n)$, which are the spaces of the query plans of the queries $Q_0, Q_1, \cdots Q_n$, respectively.

These two levels of the search space are illustrated in *Fig. 3.1.* The solid lines represents the edges at level_1 search space, and the dotted triangles together constitute the level_2 search space.

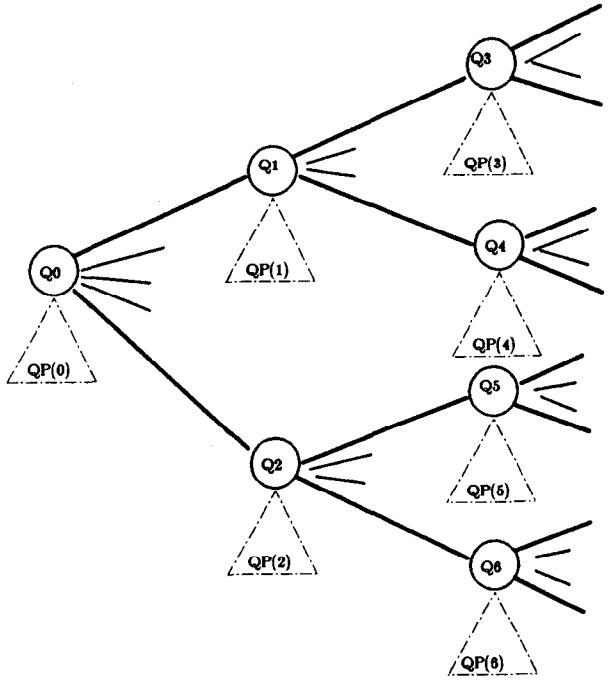

Fig. 3.1 *Conventional Vs. Semantic Query Optimization.*

Such a two level model of the search space is useful as it enables easy visualization of the search problem facing the semantic optimizer. At level 1, the semantic optimizer searches for Q_i, the semantically equivalent version of the original query Q_0 which leads to the optimal query execution plan.

Research in conventional optimization has developed efficient algorithms for searching in each of the spaces, $QP(i)$, of Level 2. Our aim is to design an algorithm for the search in Level 1 which, when at a node Q_i of Level 1, calls a standard Level 2 search algorithm as a subroutine to search the space $QP(i)$. We later indicate improvements to this basic scheme.

3.2. Searching Techniques

Semantic query optimization can be viewed as the search for the minimum cost query execution plan in the space of all possible execution plans of the various semantically equivalent but syntactically different versions of the original query. The need for a suitable search strategy to guide the search through the space of semantically equivalent queries was addressed earlier. Search techniques have been studied extensively in the field of artificial intelligence [9,10] and can be divided into two broad classes, as described below.

Blind Searching: This class of search techniques does not utilize any specific knowledge or properties about the search space. It requires the ability to recognize a solution to enable the search to be stopped. Traversing the entire search space † guarantees the discovery of the optimal solution. This generality makes it powerful enough to be applied to a very broad class of problems, but at the same time makes it a very expensive and inefficient way of searching since no domain knowledge about the problem instance is used. *Depth First Search (DFS)* and *Breadth First Search (BFS)* are examples of this class. DFS may take very long to execute if it does not traverse the search space in the right direction. BFS has prohibitive storage requirements for its execution. Recently a hybrid scheme *Depth First Iterative Deepening (DFID)* [11] has been proposed, which avoids the pitfalls of both DFS and BFS.

Heuristic searching: These search techniques utilize problem specific information as heuristics for guiding the traversal of the search space. For general or complex problem spaces, such heuristic based search techniques are almost always more efficient and certainly more interesting. Best first searches are a subset of heuristic search techniques which are very popular in artificial intelligence. The A^* algorithm [12,13] is a well known example of a best first search technique . Best first searches involve traversing the most *promising* path at any node in the search space. An appropriate heuristic function is used to compute the *promise* of a path. Best first searches combine the advantages of heuristics with other blind search techniques like DFS and BFS ‡. If the goal

† for finite search spaces

‡ Best first searches cause some depth first search at the most promising node and if a solution is not found, this node soon becomes less promising as compared to some other as yet unexplored node which is then expanded and subsequently explored.

(optimal solution) is defined precisely and its distance from the current step can be characterized by an appropriate heuristic function, A^* guarantees the discovery of the optimal solution.

3.3. Searching Technique for Semantic Query Optimization

For a semantic query optimizer, each node of the first level of the search space(see Fig 3.1) corresponds to a semantically equivalent version of the original query. The *promise* of a node is the cost of the most efficient execution plan (obtained from a conventional query optimizer) for the query corresponding to that node. At each step, the most *promising* node among those generated so far is selected for expansion. From the chosen node, various other nodes(semantically equivalent queries) are generated by applying the various possible semantic transformations on the query corresponding to the current node.

It is important to note that in viewing semantic query optimization as a search for the optimal query plan (i.e., the plan with the least total execution cost), we do not have *a priori* knowledge about the *goal*, i.e. the optimal query plan. Hence, at any step of the search, the distance to the goal (i.e. the estimated cost of reaching the goal) cannot be characterized accurately . This makes it difficult to use heuristic based search algorithms like A^*. Blind search techniques like DFS and BFS can still be applied since they are basically exhaustive enumeration methods. However, the enormously large size of the search space of possible query plans makes it impractical to do so.

Having ruled out the possibility of achieving the minimum total cost execution plan either by heuristics (required information not available) or by blind searching (very large size of search space), we focus our attention on execution plans that are *near-optimal*. There can be many ways of defining a near-optimal query execution plan. Our approach is to maintain a balance between the total cost of query optimization and the cost of the best execution plan, at any time and to stop the search when certain *stopping conditions* hold. This integrated approach, which considers both query optimization and query execution costs, is based on the following observations.

[1] The search space is far too large for exhaustive enumeration, and the knowledge required for heuristics that guarantee optimality is not present. There needs to be some criterion to stop the search after a certain time.

[2] Currently optimization cost is very small compared to execution cost of the query plan obtained. With an order of magnitude increase in the size of the space of query plans, and thus the enormous potential for further optimization, the effort spent in doing so should be a fraction of the cost of best execution plan. This is especially helpful for *compile-and-store* queries.

3.4. Termination Criteria

A search algorithm usually terminates by reaching its goal node, which satisfies the objectives. Our goal of semantic query optimization is to minimize the total *query evaluation cost*, Since the goal cannot be characterized in terms of the available information from partial search, it is not possible to stop the search at the goal node. However, we can characterize a set of nodes which are sufficiently close to the goal node and stop the search on reaching one of them.

It is possible to characterize the goal under some utility theoretic assumptions like diminishing marginal return[14]. We can define stopping criteria to minimize the total query evaluation cost under these assumptions about the search space. However, it seems very difficult to do so for general search spaces.

4. Traversing the Search Space

At any step in the search, the set *Boundary_Nodes* contains the nodes from which the next one to be visited is selected. Search space traversal has to address three problems. The first problem is to compute, at any step of the algorithm, the promise of all the *child*† nodes. The second problem is to select the *most promising* node, from amongst the boundary nodes, as the one to be visited next. The third problem is of deciding when to terminate the search space traversal. This section discusses these problems and some solutions for them.

4.1. Estimating the Promise of Child Nodes

The promise of a node is the potential reduction in the estimated query processing cost achievable by considering the semantically equivalent query corresponding to it. At any time during the search there exists a set of nodes that have been visited and a set of nodes that haven't been. Let Q_i be the node currently being visited, as shown in *Fig. 4.1*, and $Q_{i_1}, Q_{i_2} \cdots Q_{i_k}$ be its children. These semantically equivalent queries are created by using the rules applicable to Q_i, each being generated by applying exactly one rule.

4.1.1. Cost Estimation by Running Conventional Optimizer

Assume the search has reached node Q_i, as shown in *Fig. 4.1*. The promise of child nodes, Q_{i_j}, $1 \leq j \leq k$, is obtained by generating each one using the relevant rule to Q_i, and then running the conventional query optimizer on it. The costs for doing so are,

$$C_O(i_j) = C_O^r(i_j) + C_O^e(i_j) \quad 1 \leq j \leq k$$

This approach gives very accurate estimates of $C_E(i_j)$, $1 \leq j \leq k$, since the conventional optimizer has a very good cost model of the database, and explores all possible execution plans. However, the actual cost of running the optimizer k times may be quite high.

4.1.2. Algebraic Estimation of $C_E(i_j)$ from $C_E(i)$

Instead of running the conventional query optimizer on each child node, a simpler version of the cost model of a conventional query optimizer (eg. R^*) may be used directly to obtain rough estimates for guiding the search. Each child node, Q_{i_j}, $1 \leq j \leq k$, is compared with the Q_i. The nature of

† The unvisited nodes that have edges connecting them to the node currently being visited are *child nodes*.

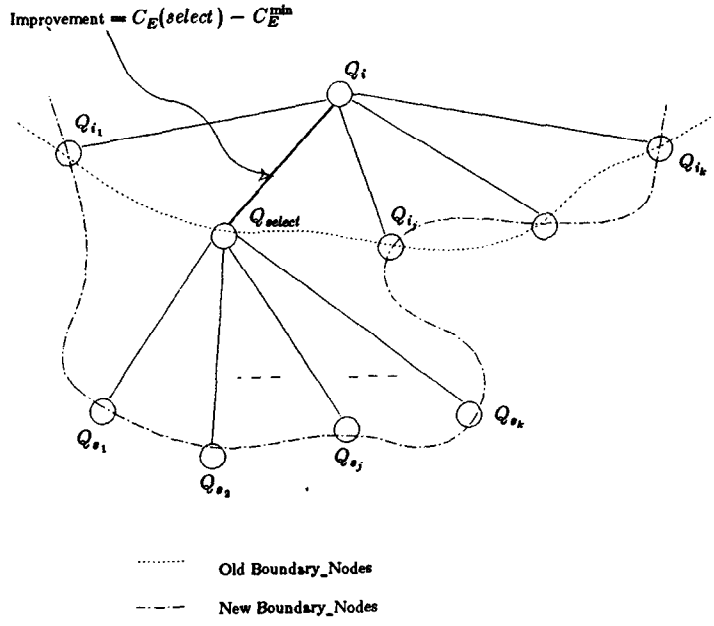

Fig. 4.1 *Selecting the Most Promising Node.*

the transformation required to obtain each from Q_i yields an estimate of the former's execution cost, which can be quite accurate in certain cases. This strategy is illustrated by an example.

Example:

Assume,

- *EMP* = (*id#*, *name*, *rank*, *age*, *salary*) be an employee relation.
- *(EMP.rank = 'manager') => (EMP.age >= 45)* be an integrity constraint satisfied by this relation.
- the attribute *age* have a clustered index of depth 3.
- the relation have P data pages.
- tuples are uniformly distributed over age between 20 and 60.
- *retrieve (EMP.name, EMP.age, EMP.salary) where (EMP.rank = 'manager')*

is a query, say Q_0.

Assuming a single disk I/O to be the unit of cost, it costs P to execute Q_0. However, Q_0 can be transformed into another semantically equivalent version, say Q_1, by applying the integrity constraint shown above.

Now,

- *retrieve (EMP.name, EMP.age, EMP.salary) where (EMP.rank = 'manager') and (EMP.age >= 45)*

is Q_1.

Q_0 and Q_1 are semantically equivalent since they produce the same result under all instances of the relation that satisfy the integrity constraint. The clustered index on *age* can be used to execute Q_1 efficiently, the cost of which would be $(\frac{60-45}{60-20})P + 3 = (\frac{15}{40})P + 3 = 0.375P + 3$ units.

Such an analysis is limited in its applicability, since it often requires knowledge about the details of the database organization. However, we include it here since it is a very powerful tool when applicable, especially since estimating the promise of child nodes costs less since the optimizer does not have to be run on each of the child nodes (queries). This approach provides a rough-and-ready rule of thumb.

4.1.3. Simultaneous Query Optimization

A third approach for estimating the promise of the various child nodes of Q_i is to use *simultaneous query optimization*. This requires enhancing the query optimizer to handle the simultaneous optimization of a set of similar queries. The set of queries, Q_{i_j}, $1 \leq j \leq k$, which are child queries reachable from Q_i, are all input to the optimizer at the same time and optimized together. Since these queries resemble each other very closely, in fact differ from Q_i by exactly one clause each, their simultaneous optimization will lead to substantial savings. As was observed in our experiments, many semantically equivalent queries that differed only slightly produced many execution plans that were the identical, and had to be produced once for each query since each was optimized separately. The approach of simultaneous query optimization will lead to each such plan being generated exactly once for all the queries optimized together. The existing optimizers, eg. *system-R* [4], *INGRES* [5], *R** [15], etc. need to be enhanced to allow this. Research in the areas of *global query optimization* [16,17], and *common subexpression analysis* [18] can be used for this purpose.

4.2. Choice of the Most Promising Candidate Node

Once the promise of each candidate node has been obtained, by one of the methods described above or some

other method, they are compared to select the *most promising* one. There can be many ways of doing this, depending on how the promises are compared. Described below is a well known strategy.

Moving from query Q_i to Q_j change the estimated best query execution cost, as shown in *Fig. 4.1.* The change can be an increase or decrease, leading to a more efficient semantically equivalent query or a less efficient one. The next node to visit, Q_{select}, is determined as follows,

$$Q_{select} = Q_j \quad such\ that \quad C_E(j) = \min_{Q_l \in Boundary\_Nodes} \{C_E(l)\}$$

This approach is called the *First-Order Best First Search* strategy, in which the most promising node is the one that has the potential of maximum cost reduction. The node Q_{select} is removed from *Boundary_Nodes* and its child nodes, $Q_{s_1}, Q_{s_2} \cdots Q_{s_k}$ are added to it.

Thus,

Boundary_Nodes:=

$$Boundary\_Nodes - \{Q_{select}\} \bigcup \{Q_{s_1}, Q_{s_2} \cdots Q_{s_k}\}$$

This is a *First-Order* strategy because in evaluating the promise of a direction to move in, only the first node in that path is examined. A generalized method may look at some k nodes on a path before taking a step. In our case the *first-order* strategy yielded quite satisfactory results.

4.3. Terminating the Search

Searching theory in Artificial Intelligence (AI) usually ignores the cost of searching and concentrates only on the quality of the solution, i.e. how it compares with the optimal one. For SQO, we have to consider the trade-off between the cost of optimization and solution quality (i.e. query execution time). There are two reasons for this: *(i)* the time required for exhaustive search of the space of query plans for the entire set of semantically equivalent queries can be prohibitively large, and *(ii)* the response time for interactive queries depends on both the query execution time and the query optimization time.†

As stated earlier there does not exist an *a priori* characterization of the query, in the set of semantically equivalent queries, that leads to the optimal (i.e. minimum cost) query plan. Thus we have to use some *stopping criteria* to terminate the search after a reasonable amount of effort has been spent.

4.3.1. Criterion C1: Balancing Optimization Cost with Execution Cost

An intuitively appealing stopping rule is based on balancing the total optimization cost incurred at any step of the search, with the estimated execution cost of the best query plan found so far. Searching stops after the step in which the total optimization cost incurred so far reaches a certain fraction of execution cost of the best query plan yet discovered. Surprisingly enough, this simple rule gives us a good bound on the response time of the query, as shown by *Theorem 1.* A point to note is that parameter for optimization (minimization in our case) is the *response time* $(\tau + t)$ and not *execution time* (t) alone. It is entirely possible that the point at which the response time is minimized may be different from the one that minimizes the execution time. This fact is illustrated in *Fig. 1.1.* We introduce some notation in *Table 4.1* which is used in the proof below.

Symbol	Meaning
$SP(i)$	space of queries already explored upto and including Q_i
$\tau(i)$	total optimization cost upto node Q_i, i.e. $\sum_{Q_j \in SP(i)} C_O(j)$
$t(i)$	cost of best query plan so far, i.e $\min_{Q_j \in SP(i)} C_E(j)$
$RT(i)$	response time if search terminates at Q_i,i.e. $\tau(i) + t(i)$
$RT(opt)$	theoretically minimum response time possible; obtainable only from an Oracle $RT(opt) = \min_{0 \le i \le n} RT(i)$
$\tau(opt)$, $t(opt)$	components of $RT(opt)$
Q_{opt}	node, i.e. query, corresponding to $RT(opt)$
α	average number of times a compile-and-store query is executed

Table 4.1 *Searching Related Notation.*

Theorem 1: If the following search terminating criterion $(C1)$ is used,

$$\sum_{Q_j \in SP(i)} C_O(j) \ge \frac{\min_{Q_j \in SP(i)} C_E(j)}{\lambda}$$

$$\text{i.e. } \tau(i) \ge \frac{t(i)}{\lambda},$$

the following upper bound on $RT(i)$ is obtained,

$$\frac{RT(i)}{RT(opt)} \le max(1 + \lambda, 1 + \frac{1}{\lambda})$$

Proof: Supposing the search terminates after examining Q_i. There are following two cases,

[a] $Q_{opt} \in SP(i)$ i.e. Q_{opt} has been visited.

[b] $Q_{opt} \notin SP(i)$ i.e. Q_{opt} hasn't been visited.

Case [a]:

$$\frac{RT(i)}{RT(opt)} = \frac{\tau(i) + t(i)}{\tau(opt) + t(opt)} = \frac{\frac{t(i)}{\lambda} + t(i) + \delta}{\tau(opt) + t(opt)} \quad \text{since } \tau(i) = \frac{t(i)}{\lambda} + \delta$$

A positive quantity δ is added to the numerator to make the equality hold. It satisfies the condition $0 \le \delta \le$ *Opt cost in current step.*

$$\Rightarrow \frac{RT(i)}{RT(opt)} \le \frac{\frac{t(i)}{\lambda} + t(i) + \delta}{t(opt)} \quad \text{since } \tau(opt) \ge 0$$

$$\le \frac{(\frac{1}{\lambda} + 1)t(opt) + \delta}{t(opt)}$$

because $Q_{opt} \in SP(i) \Rightarrow t(i) \le t(opt)$ †

† In reality, *Response_Time = Optimization_Time + Execution_Time + Queuing Delays*. We do not consider the last component here.

† Note that $t(opt)$ is the estimated query execution cost corresponding to the query that gives $RT(opt)$, and not the least estimated query execution cost. See *Fig. 1.1.*

$$\approx 1 + \frac{1}{\lambda} \qquad \text{Opt cost in current step} \ll t(opt) \Rightarrow \frac{\delta}{t(opt)} \approx 0 \ddagger$$

Case [b]:

$$\frac{RT(i)}{RT(opt)} = \frac{r(i) + t(i)}{r(opt) + t(opt)}$$
$$\leq \frac{r(i) + \lambda r(i)}{r(opt) + t(opt)} \qquad r(i) \geq \frac{t(i)}{\lambda}$$
$$\leq \frac{r(i) + \lambda r(i)}{r(opt)} \qquad t(opt) \geq 0$$
$$\leq \frac{(1+\lambda) r(i)}{r(i)} \qquad Q_{opt} \notin SP(i) \Rightarrow r(opt) \geq r(i)$$
$$= 1 + \lambda$$

Combining the results of the cases above we have,

$$\frac{RT(i)}{RT(opt)} \leq max(1 + \lambda, 1 + \frac{1}{\lambda})$$

The bound provided by the above theorem is not very tight, and the termination criterion performs much better in practice. This was observed when we ran an experiment, details of which are discussed in section 5.

The choice of λ depends on the size of the search space, and the time of query optimization. For a small search space, we would usually expect *Case [a]* to occur when the search terminates, i.e. $Q_{opt} \in SP(i)$. Thus,

Small Space $\approx>$ Case [a]
$\Rightarrow$ choose large λ for tight bound
$\Rightarrow$ less optimization is good for response time

For a large search space, we would usually expect *Case [b]* to occur when the search terminates, i.e. $Q_{opt} \notin SP(i)$. Thus,

Large Space $\approx>$ Case [b]
$\Rightarrow$ choose small λ for tight bound
$\Rightarrow$ more optimization is good for response time

Rule of Thumb for choosing λ: Above analysis shows that the thumb-rule is to choose a large λ if the search space is small and *vice versa*. It is not a certain rule because of the approximate implications ($\approx>$) shown above.

Corollary 1: If the query Q_0 is the *compile–and–store* type, and is expected to be executed α times, the above bound changes to,

$$\frac{RT(i)}{RT(opt)} \leq max(1 + \alpha\lambda, 1 + \frac{1}{\alpha\lambda}) \dagger$$

Proof: Identical to that of *Theorem 1.*

4.3.2. Criterion C2: Diminishing Marginal Returns

Searching strategies like *best-first* usually have diminishing marginal returns as the search progresses. This is illustrated in the example discussed in Section 5. Database access paths are tailored to suit the frequently occurring queries, and semantic transformation based on integrity constraints can not keep improving the execution cost for a long time. The optimal query Q_{min}‡ has a finite positive cost, and as we approach it during the search, the chances of substantial improvement in query execution cost keep diminishing. At the same time the optimization cost remains approximately the same for every step of searching in the space of queries. Thus, net improvement in query execution cost keeps diminishing.

A termination criterion based on diminishing marginal returns considers the net benefit obtained from the last step. The space of semantically equivalent queries is explored till no transformation causes a decrease in total cost, i.e. $RT(i)$. Search stops when the optimization cost in last step dominates the improvement in query execution cost. Thus the termination criterion *C2* is,

$$\{ \min_{Q_j \in SP(i) - \{Q_i\}} C_E(j) \} - C_E(i) \leq C_O(i)$$

This termination criterion leads to the optimal solution only if the law of diminishing marginal utility holds. However, even if this is not true, the criterion is still useful for small search spaces characteristic of most queries posed to the database. The searching process may be terminated at a local minima of the query execution cost, $C_E(i)$. This can be partially overcome by using probabilistic search guiding strategies, eg. *simulated annealing*[19,20].

5. A Detailed Example

This section discusses an experiment to test our search algorithm. Semantic query optimization of a rather elaborate query was carried out. The query optimizer of R^*[15], was used for estimating the promise of queries.

5.1. Parameters of the Experiment

A shipping database of six relations reported in[6] was used. The database schema, the relation sizes, and the various indexes available are shown in *Table 5.1.*

Fig. 5.1 gives the rules that define a subset of the semantic integrity constraints satisfied by the database, applicable to the example query Q_0.

The unique logical access paths among the relations are specified. The implicit joins that underlie the logical access paths are listed in *Fig. 5.2.*

OWNERS.OwnerName = SHIPS.Owner
SHIPS.ShipName = CARGOES.Ship
PORTS.PortName = CARGOES.Destination
CARGOES.Insurance = POLICIES.Policy
POLICIES.Issuer = INSURERS.Insurer

Fig. 5.2 *Unique Join Predicates.*

Finally, the query, expressed in a Prolog like syntax, is,

Q_0: (DeadWt > 400) and (DollarValue > 4000): (?Destination)

In running our experiment on R^* we did not create the entire database. Instead we simulated its presence by inserting appropriate parameters in the *system catalog*, which is the source of information for the optimizer.

5.2. Space of Semantically Equivalent Queries

The first step in our experiment was to generate the entire space of semantically equivalent queries. This was done manually by applying rules R_1 through R_8 to Q_0. Actually

---

† $RT(i) = r(i) + \alpha t(i)$, and represents an integrated cost.

‡ This assumption is justified by most conventional query optimizers, and is thus not unrealistic.

‡ $Q_{min} = Q_i$ *such that* $C_E(i) \leq C_E(j)$, $j \neq i$, $j = 0, 1, \cdots n$

Relation	Attributes	Attribute for Primary Index (Clustered)	Attributes for Secondary Indices (Unclustered)	Relation Size (in tuples)
SHIPS	ShipName, Owner, ShipType, Draft, DeadWt, Capacity, Registry	ShipName	ShipType, Owner, DeadWt	20,000
PORTS	PortName, Country, Depth FacilityType	PortName	---	1,000
CARGOES	Ship, Destination, Shipper CargoType, Quantity, DollarValue, Insurance	Ship	Destination, Insurance	25,000
OWNERS	OwnerName, Location, Assets, Business	OwnerName	Business	1,000
POLICIES	Policy, Issuer, Coverage	Policy	Issuer	25,000
INSURERS	Insurer, InsCountry, Capitalization	Insurer	---	500

Table 5.1 *The Shipping Database.*

Rule #	Antecedent	Implication	Consequent
R_1	*(DeadWt > 950)*	-->	*(FacilityType = 'OffShore')*
R_2	*(DeadWt > 800)*	-->	*(Business = 'Leasing')*
R_3	--	--	*(Coverage <= DollarValue)*
R_4	--	--	*(Quantity <= Capacity)*
R_5	*(CargoType* ϵ {'Natural Gas', 'Refined'}) and *(DollarValue > 500)*	-->	*(Facilitytype = 'General')*
R_6	*(DeadWt > 150)*	-->	*(ShipType = 'SuperTanker')*
R_7	*(DollarValue > 9000) and (ShipType = 'SuperTanker')*	--->	*(Issuer = 'Lloyds')*
R_8	*(Business = 'Petroleum')*	-->	*(CargoType* ϵ {'Natural Gas', 'Refined', 'Oil'})

Fig. 5.1 *Semantic Integrity Constraints.*

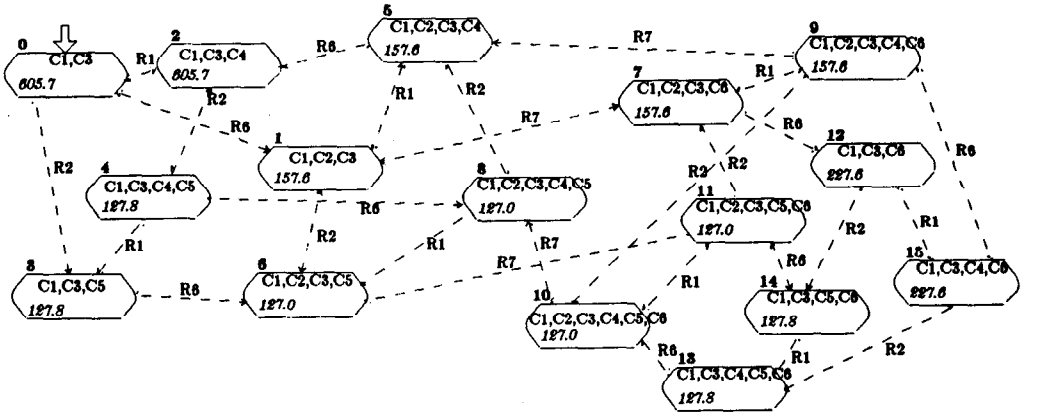

Fig. 5.3 *Space of Semantically Equivalent Queries.*

only rules R_1, R_2, R_6 and R_7 were applicable, giving rise to queries Q_1 through Q_{15}, which were all semantically equivalent to Q_0.

Fig. 5.3 shows the entire space generated, in which each node represents a semantically equivalent but syntactically distinct query. The clauses C_1 through C_6 are as follows,

C_1: *(DeadWt > 400)*
C_2: *(ShipType = 'SuperTanker')*
C_3: *(DollarValue > 4000)*
C_4: *(FacilityType = 'OffShore')*
C_5: *(Business = 'Leasing')*
C_6: *(Insurer = 'Lloyds')*

The transitions between nodes in *Fig. 5.3* represent the transformations carried out, by rule applications. Two basic inference rules, called ***clause introduction (CI)*** and ***clause elimination (CE)*** are used to carry out the transformations.

Let,
A,B be distinct clauses,
and, *(A --> B)* be a rule.

Now,
(CI): (A) and (A --> B) $\equiv$ *(A) and (B) and (A --> B)*
(CE): (A) and (B) and (A --> B) $\equiv$ *(A) and (A --> B)*

```
procedure Semantic_Search;
begin */ initialization /*
      C_E^min := ∞;
      Boundary_Nodes:= {Q_0};
      Run a conventional optimizer to evaluate C_E(0);
      loop */ terminate if Boundary_Nodes is empty /*
        exit if Boundary_Nodes = φ;
            */ choose most promising node from Boundary_Nodes /*
            Q_select = {Q_j | (C_E(j) < C_E(l))} j ≠ l, Q_j, Q_l ∈ Boundary_Nodes;
            */ if better plan found then update C_E^min /*
            if C_E^min > C_E(select) then
            C_E^min:= C_E(select);
    X:  exit if Termination Criterion = true;
            */ remove Q_select from Boundary_Nodes /*
            Boundary_Nodes:= Boundary_Nodes − {Q_select};
            */ add Q_select's children, Q_s1, Q_s2 ··· Q_sk to
            Boundary_Nodes /*
    Y:          Boundary_Nodes:= Boundary_Nodes ⋃ {Q_s1, Q_s2 ··· Q_sk};
            */ estimate the promise of the newly added nodes /*
            for i:= s_1, s_2, ··· s_k do
     Z:     Run a conventional optimizer to evaluate C_E(i);
        endloop;
end;
```

Fig. 5.4 *The Search Algorithm for SQO.*

Rules like *(A --> B)* are integrity constraints of the system are all always true. Thus we are only interested in the clauses *A, B, A and B,* etc.

5.3. Data Collection

Each of the queries Q_0 through Q_{15} was entered into the system in an interactive manner using SQL, and optimized by the R^* optimizer. The optimizer generates a number of query execution strategies (called query plans) and then attaches an estimated cost with each by using a cost model based on the the expected number of CPU instructions and page fetches. A detailed discussion of the cost model is given in[15]. The execution cost estimates provided by the optimizer are dependent on the system configuration and thus we do not give any units. An elaborate description of the configuration and parameters assumed can be found in[15]. However, for our present purpose only their relative values are important. The data collected is shown in *Table 5.2.*

Query#	0	1	2	3	4	5	6	7
Total Cost	605.7	157.6	605.7	127.8	127.8	157.6	127.0	157.6
Plans Costed	10	10	28	28	58	28	29	57
Query#	8	9	10	11	12	3	14	15
Total Cost	127.0	157.6	127.0	127.0	227.6	127.8	127.8	227.6
Plans Costed	59	142	250	102	57	249	101	142

Table 5.2 *Statistics from the Query Optimizer.*

5.4. Description of the Search Algorithm

A description of our search algorithm is given below. Q_i's are used interchangeably for queries/ nodes.

A further optimization is possible at statement *Z* of *Fig. 5.4* above. All the different possible semantic transforms possible from a given node will not necessarily lead to query plans with lower execution costs. It is often possible to identify with the help of a few heuristics, those semantic transformations which may lead to reductions in query execution time. By only utilizing the *useful* semantic transforms at a given node, the computational efficiency of procedure *semantic search* can be improved. This process of pruning the search space is termed as *semantic pruning.*

5.5. Execution of the Search Algorithm

The search algorithm described previously was hand-simulated on the graph shown in *Fig. 5.3.* The execution trace is shown in *Table 5.3.* An iteration of the algorithm is completed when the condition in statement *X* of *Fig. 5.4* is tested. Row *i* of the table corresponds to the state of the search algorithm *after* iteration *i* has just been completed. The costs included in *Table 5.3* are only of the statements *Y* and *Z* of *Fig. 5.4,* since other steps of the algorithm have negligible cost.

All costs shown are in the same units, say milliseconds. The parameters p and r are defined as follows.

p: Average cost of a single plan evaluation by the optimizer.

r: Average cost of traversing an arc in *Fig. 5.3,* via an application of rule *CA* or *CE.*

The values assigned to these parameters were $p = 1$ *ms* and $r = 5$ *ms*, assuming a 10 MIPS machine. A simple calculation shows that this corresponds to 10,000 m/c instructions for evaluation of an average plan, and 50,000 m/c instructions for an average arc traversal (logical inference). The derivation of the value of r also assumes that there are 100 rules in the database, which corresponds to 500 instructions on the average for checking each rule, which involves patterns matching and some other work. These assumptions are quite reasonable according to current technology.

Two criteria for stopping the search algorithm were discussed above. We illustrate those in the context of the example, in following subsections.

Iteration #	Current Node Q_i	Boundary Nodes	Nodes Evaluated in Current Step	Optimization Cost in Current Step	Optimization Cost so far $\tau(i)$	Minimum exec cost, C_E^{min}	Q_{select}
1	-	{0}	{0}	10p	10p	605.7	0
2	0	{1,2,3}	{1,2,3}	66p + 3r	76p + 3r	127.8	3
3	3	{1,2,4,6}	{4,6}	87p + 2r	163p + 5r	127.0	6
4	6	{1,2,4,8,11}	{8,11}	161p + 2r	324p + 7r	127.0	8
5	8	{1,2,4,5,10,11}	{5,10}	278p + 2r	602p + 9r	127.0	{10,11}

Table 5.3. *Execution of Search Algorithm.*

5.5.1. Termination Criterion C1

The first stopping rule terminates the search when the following condition becomes true.

$$\tau(i) > \frac{t(i)}{\lambda}$$

The stopping rule was examined for the values $\lambda = 2$, 1, and $\frac{1}{2}$. The results are presented in *Table 5.4*.

Iteration #	$\tau(i)$	$t(i) \equiv C_E^{min}$	$\lambda=2$ $\tau(i)+t(i)$,	$\lambda=1$ $\tau(i)+t(i)$,	$\lambda=\frac{1}{2}$ $\tau(i)+t(i)$
1	10	605.7	615.7	615.7	615.7
2	97	127.8	**224.8**	224.8	224.8
3	188	127.0	315.0	**315.0**	315.0
4	359	127.0	486.0	486.0	**486.0**
5	647	127.0	774.0	774.0	774.0

Table 5.4 *Performance of Criterion C1.*

$\lambda = 2$: The search stops after iteration 2 as shown in *Table 5.4*, since *C1* evaluates to $(76)(1) + (3)(5) > \frac{127.8}{2} \Rightarrow 91 > 63.9$, which is true. The optimization cost and best execution cost estimate are,

$\tau(i) = 91\ ms$; $t(i) = 127.8\ ms$

The bound of *Theorem 1* is satisfied since,

$$\frac{\tau(i) + t(i)}{\tau(opt) + t(opt)} = \frac{224.8}{224.8} \leq 3 = \max(1 + 2, 1 + \frac{1}{2})$$

$\lambda = 1$: The search stops after iteration 3, since *C1* evaluates to $(163)(1) + (5)(5) > 127.0 \Rightarrow 188 > 127.0$, which is true. The optimization cost and best execution cost estimate are,

$\tau(i) = 188\ ms$; $t(i) = 127.0\ ms$

The bound of *Theorem 1* is satisfied since,

$$\frac{\tau(i) + t(i)}{\tau(opt) + t(opt)} = \frac{315.0}{224.8} \leq 2 = \max(1 + 1, 1 + \frac{1}{1})$$

$\lambda = \frac{1}{2}$: The search stops after iteration 4, since *C1* evaluates to $(324)(1) + (7)(5) > \frac{127.0}{1/2} \Rightarrow 359 > 254.0$, which is true. The optimization cost and best execution cost estimate are,

$\tau(i) = 359\ ms$; $t(i) = 127.0\ ms$

The bound of *Theorem 1* is satisfied since,

$$\frac{\tau(i) + t(i)}{\tau(opt) + t(opt)} = \frac{224.8}{224.8} \leq 3 = \max(1 + \frac{1}{2}, 1 + \frac{1}{1/2})$$

Thus we can see that stopping rule 1 is quite effective. Especially notable is the fact that even though we attempt to minimize a weighted sum of $\tau(i)$ and $t(i)$, and not $t(i)$ alone, the value of $t(i)$ obtained is actually quite close to the minimum. We believe a more careful analysis of the algorithm is required to explain this.

5.5.2. Termination Criterion C2

The second stopping rule terminates the search when the following condition is satisfied at any step,

Reduction in $t(i)$ < *Optimization Cost*

At step 1 we have,

Reduction in $t(i)$ *in step* 1
$= \infty - 605.7 = \infty\ ms$
Optimization Cost in step 1
$= (10)(1) = 10\ ms$

At step 2,

Reduction in $t(i)$ *in step* 2
$= 605.7 - 127.8 = 477.9\ ms$
Optimization Cost in step 2
$= (66)(1) + (3)(5) = 81\ ms$

At step 3,

Reduction in $t(i)$ *in step* 3
$= 127.8 - 127.0 = 0.8\ ms$
Optimization Cost in step 3
$= (87)(1) + (2)(5) = 97\ ms$

Thus, the search stops after step 3, having incurred a total optimization cost of 188 *ms*, and having generated a query execution plan with an estimated cost of 127.0 *ms*. The above analysis shows that stopping rule 2 is effective, since even for an interactive query (i.e. execute only once) the savings obtained are $605.7 - (188.0 + 127.0) = 605.7 - 315.0 = 290.7\ ms$. If the query is to be executed many times the savings are even greater.

6. Conclusions

The search space of execution plans for queries involving a many relations becomes large enough to make the optimization cost comparable to the execution cost. This problem is exacerbated during semantic query optimization, an approach to query optimization which is gaining popularity due to its potential for improvements beyond conventional methods. For interactive queries especially, the objective function to minimize is no more the execution time, but rather the response time, i.e. the sum of optimization and execution times. Since there exists no characterization for the optimal solution, classical heuristic search techniques are inapplicable. We have presented a best-first heuristic search algorithm with termination criteria based on utility theory, and derived an upper bound on the quality of solution it produces. Experimental evidence shows that in reality the algorithm does quite well.

Many issues and problems need to be resolved. Our bound for near-optimality of the tradeoff between semantic optimization quality of execution plan is not tight. In practice our stopping criteria seem to perform much better, and we believe that the bound can be improved. It is non-trivial to design an optimal search algorithm to minimize total query evaluation cost, without exploring the entire search space. It would be interesting to characterize special cases of the search space, for which optimal search algorithms are possible. Finally, performance of heuristic search based semantic query optimization needs to be evaluated in a real database environment.

There are other ways of improving performance of query optimizers, and research efforts also need to be directed towards better modeling of random events, underlying database organization and compile time events[21].

7. Acknowledgements

We wish to thank Dr. Guy Lohman for his help with the R^* optimizer, which was used for experimental validation. We'd also like to thank to Prof. Mike Stonebraker for suggesting this problem, and to Prof. Richard Korf for fruitful discussion about utility theory. We also thank IBM for for letting us use their facilities at Almaden Research Center for experimental work.

References

1. Stonebraker, M., *Private Communications*, Summer 1987.
2. Smith and Chang, "Optimizing the performance of a relational algebra database interface," *CACM*, vol. 18:10, 1975.
3. Wong,E. and Youseffi,K., "Decomposition - A strategy for query optimization," *ACM TODS*, Sept. 1976.
4. P.Selinger, "Access path selection in a Relational Database Management System," *RJ 2883*, IBM San Jose, 1979.
5. Hammer and Zdonik, "Knowledge Based query processing," *Proc. 6th Conf. on VLDB* , 1980.
6. King,J.J, "QUIST : A system for semantic query optimization in relational databases," *Proc. 7th VLDB Conf.*, 1981.
7. S.T.Shenoy and Z.M.Ozsoyoglu, "A System for Semantic Query Optimization," *Proc. ACM-SIGMOD* , pp. 181-195, 1987.
8. U.S.Chakravarthy, et.al., "Semantic query optimization in expert systems and database systems," *Proc. Expert Database Systems Conf.*, pp. 326-341, 1984.
9. E. Rich, *Artificial Intelligence*, McGraw-Hill, New York, 1983.
10. N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.
11. Richard E. Korf, "Depth-First Iterative Deepening : An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.
12. P.E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Sci. Cybernet.*, vol. 4(2), pp. 100-107, 1968.
13. R. Detcher and J. Pearl, *Generalized best-first strategies and the optimality of A**, UCLA-ENG-8219, University of California, Los Angeles, 1983.
14. W. Jacobs and M. Kiefer, "Robot Decisions based on Maximizing Utility," *Proc. of 3rd IJCAI*, 1973.
15. L. F. Mackert and G.M.Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proc. ACM-SIGMOD*, pp. 84-95, ACM, 1986.
16. T.Sellis, "Global Query Optimization," *Proc. ACM-SIGMOD Conf.*, 1986.
17. J. Park, "Multiple Query Optimization (prepublication)," *Graduate School of Bus. Adm.*, Univ. of Cal., Berkeley, 1987.
18. S.Finkelstein, "Common Expression Analysis in Database Applications," *Proc. ACM-SIGMOD* , 1982.
19. Ioannidis, Y.E. and Wong,E., "Query Optimization by Simulated Annealing," *Proc. SIGMOD*, 1987.
20. Kirkpatrick, S. et.al., *Optimization by Simulated Annealing*, 220, pp. 671-680, May 1983.
21. G.M.Lohman, "Panel Discussion on Semantic Query Optimization," *Proc. Data Engineering Conf.*, 1985.