QUERY OPTIMIZATION BY SIMULATED ANNEALING

by

Yannis E. Ioannidis
Eugene Wong

Computer Sciences Technical Report #693

April 1987

QUERY OPTIMIZATION BY SIMULATED ANNEALING

Yannis E. Ioannidis
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Eugene Wong
Electrical Engineering and Computer Sciences Department
University of California
Berkeley, CA 94720

(To appear in Proc. of SIGMOD '87)

This research was supported by the National Science Foundation under Grant ECS-8300463.

QUERY OPTIMIZATION BY SIMULATED ANNEALING [1]

Yannis E. Ioannidis [2]
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Eugene Wong
Electrical Engineering and Computer Sciences Department
University of California
Berkeley, CA 94720

Abstract

Query optimizers of future database management systems are likely to face large access plan spaces in their task. Exhaustively searching such access plan spaces is unacceptable. We propose a query optimization algorithm based on *simulated annealing*, which is a probabilistic hill climbing algorithm. We show the specific formulation of the algorithm for the case of optimizing complex non-recursive queries that arise in the study of linear recursion. The query answer is explicitly represented and manipulated within the *closed semiring* of linear relational operators. The optimization algorithm is applied to a state space that is constructed from the equivalent algebraic forms of the query answer. A prototype of the simulated annealing algorithm has been built and few experiments have been performed for a limited class of relational operators. Our initial experience is that, in general, the algorithm converges to processing strategies that are very close to the optimal. Moreover, the traditional processing strategies (e.g., the *semi-naive evaluation*) have been found to be, in general, suboptimal.

[1] This research was supported by the National Science Foundation under Grant ECS-8300463.

[2] This work was done when the first author was a student at the Computer Science Division at the University of California, Berkeley.

1. INTRODUCTION

The key to the success of a Database Management System (DBMS), especially of one based on the relational model [Codd70], is the effectiveness of the query optimization module of the system. The input to this module is some internal representation of an ad-hoc query. Its purpose is to select the most efficient algorithm (access plan) in order to access the relevant data and answer the query.

Query optimization has been an active area of research ever since the beginning of the development of relational DBMSs. Various query optimization algorithms have been developed as part of the research projects of INGRES [Ston76, Wong76, Kooi80] and System R [Astr76, Blas76, Seli79, Mack86b]. Theoretical approaches to query optimization have also been employed, attempting to apply general theorems to help the query optimizer in its task [Aho79, Rose80]. Query optimization algorithms in a distributed DBMS environment have received considerable attention as well [Epst78, Bern81, Mack86a]. A good survey on query optimization and other related issues can be found in the survey article by Jarke and Koch [Jark84] and the book by Kim, Reiner, and Batory [Kim86].

Most of the existing work on query optimization has focused on optimizing conjunctive queries. The demand for such queries in current database applications is higher than for queries that also involve disjunctions. In terms of relational algebra [Codd70], the three relational operators that have received almost the exclusive attention of researchers are projection, selection, and join. To the best of our knowledge, the problem of optimizing union, a necessary relational operator for disjunctive queries, has not been addressed by any previous study.

The unit of optimization in most of the existing DBMSs is a single query. Each query involves a small number of relations (e.g., less than 10). Hence, even though the number of alternative access plans to answer a query grows exponentially with the number of the relations in the query, this number is relatively small. So, most of the existing query optimizers perform an exhaustive search over the space of alternative access plans, and whenever possible, use heuristics to reduce the size of that space.

Several aspects of the picture presented above change when one studies query optimization in systems that are geared towards some of the newest database application domains, such as artificial intelligence (e.g., expert database systems [Kers86a, Kers86b] and deductive database systems [Gall78, Gall84]). The most important of these changes are identified below.

- The number of relations expected to participate in a query increases significantly [Kris86].
- The unit of optimization changes from a single query to a set of queries. In other words, the optimizer tries to optimize the execution of several queries together, possibly taking advantage of common tasks that have to be performed by more than one query. This type of optimization is referred to as *global optimization* [Gran81, Sell86].
- Queries may become recursive. Recursive queries are, in general, equivalent to the union of an arbitrary number of nonrecursive queries. In general, the number of nonrecursive queries to which a recursive one is equivalent depends on the contents of the database, and it can be arbitrarily large, independent of the simplicity of the recursive query.

Each of the above three points leads to the same conclusion: the space of access plans that the query optimizer has to face in future DBMSs is larger by several orders of magnitude than the one currently faced in conventional systems. The validity of this observation is obvious. We have already mentioned that the number of alternative access plans for a query is an exponential function of the number of relations in the query. Hence, small increases in the number of relations in a query, e.g., by one order of magnitude, result in large increases in the size of the access plan space. When performing global optimization, the state space increases at a rate higher than linear in the number of queries. This is because the optimizer does both interquery and intraquery optimization, trying to identify common subexpressions or other common characteristics in the participating queries that can speed up execution if they are taken into account. Finally, the number of nonrecursive queries that are equivalent to a recursive one is arbitrarily large. These nonrecursive queries share several common subexpressions, since each is equivalent to repeatedly applying the same query several times. Hence, the size of the resulting access plan space can be arbitrarily large as well.

The above discussion leads to the conclusion that exhaustive search of the access plan space is no longer plausible for query optimization. In the past, a limited number of approaches have been employed to deal with large access plan spaces. For queries with a large number of relations, Krishnamurthy, Boral, and Zaniolo have devised an optimization algorithm that is quadratic in the number of participating relations [Kris86]. Their approach is to look only at a small, carefully selected part of the complete access plan space. Based on some assumptions about the form of the cost function associated with each access plan,

they have devised an efficient algorithm that most of the time gives a solution close to optimal and "always avoids the worst executions" [Kris86]. In global optimization, heuristic search algorithms, such as A* [Rich83], are employed to avoid searching the entire search space [Gran81, Sell86]. Sellis has also proposed several less general techniques that are applicable when the set of optimized queries is of some specific form [Sell86]. To the best of our knowledge, no query optimization algorithms for recursion have existed until now.

In this paper, we present a probabilistic algorithm for query optimization which is suitable for large access plan spaces. The algorithm is based on the *Simulated Annealing* process, which has been used for various other optimization problems, especially problems related to global routing and cell placement for VLSI chip design [Kirk83]. Simulated annealing is a probabilistic *hill climbing* algorithm. Hill climbing algorithms are greedy algorithms. Therefore, their outcome is in general suboptimal. A hill climbing algorithm has been used for query optimization in the SDD-1 project [Bern81]. The SDD-1 algorithm incorporated several *enhancements*, i.e., heuristics that help overcome the greediness of the algorithm. To the best of our knowledge, the algorithm presented in this paper is the first probabilistic algorithm to be used in query optimization. Simulated annealing is especially well suited to optimization problems with large search spaces and with cost functions that manifest a large number of local minima. If the number of local minima is small, then a greedy approach is probably adequate. Hence, for conventional query optimization, simulated annealing is inappropriate. This is not the case, however, when DBMSs are used in new application domains. Simulated annealing appears to be a promising approach to face the new challenges of the optimizer, as created by the large size of the access plan space.

Simulated annealing is applicable to any type of query optimization, i.e., optimization of queries involving many relations, global optimization, and recursive query optimization. In this paper, however, we concentrate on some aspects of recursive query optimization. In particular, we assume that the number of non-recursive queries that are equivalent to the recursive one is known in advance. Hence, the query that is optimized is actually non-recursive. One can think of two reasons why such an approach is still useful in addressing recursive query optimization:

- Some recursive queries are bounded, i.e., they are equivalent to a fixed finite number of non-recursive queries independent of the database contents [Ioan86a, Naug86a, Sagi86].

- For unbounded recursive queries, the number of necessary non-recursive queries can be estimated using statistics maintained by the system on the database contents. Optimization can be based on this estimate.

In the sequel, we will continue using the term recursive for the queries we are interested in, although for the application of the proposed algorithm only a relevant non-recursive query (possibly some initial finite part of the recursive one) will be optimized.

Recent studies comparing the performance of various algorithms that have been devised for recursive query processing indicate that no particular algorithm is universally optimal, i.e., for all database instances [Banc86a, Banc86b, Vald86, Ioan86b]. Hence, query optimization for recursive queries becomes necessary. There have been several studies on improving the execution cost of recursive queries that are of some specific form [Ioan86a, Banc86c, Naug86b, Sagi86]. These studies, however, do not address the general optimization problem of finding the optimal access plan to answer a given query. This last problem is the one addressed in this paper.

This paper is organized as follows. In Section 2, we describe an algebraic model for the study of recursion as introduced in [Ioan86c]. We also formulate the optimization problem in the context of that algebraic model. Most of that section is taken from [Ioan86b]. In Section 3, we present some general results that allow us to somewhat reduce the size of the access plan space that the simulated annealing algorithm has to explore. Section 4 is devoted to describing optimization by simulated annealing in general terms. In Section 5, the specifics of applying simulated annealing to recursive query optimization are given. We give the exact formulation of the algorithm as well as some preliminary experimental results produced by a prototype implementation of the algorithm. Section 6 discusses the strong and weak points of the particular formulation of simulated annealing given in this paper and suggests some improvements. Finally, Section 7 gives the summary and indicates several directions for future research.

2. OPERATOR MODEL

In this paper, we use the following canonical example for recursion. Consider a database with a stored relation **father** with schema **father**(fath,son). Using **father** in the following two Horn clauses [Gall78], we define the virtual relation **ancestor** with schema **ancestor**(anc,desc):

$$\textbf{ancestor}\,(x,z) \wedge \textbf{father}\,(z,y) \rightarrow \textbf{ancestor}\,(x,y),$$

$$\textbf{father}\,(x,y) \rightarrow \textbf{ancestor}\,(x,y).$$

The first Horn clause is recursive in the sense that the relation **ancestor** appears on both the qualification and the consequent. Answering queries on recursively defined relations, such as **ancestor** above, is the problem in which we are interested.

For the purpose of this paper, we concentrate on *linear* and *immediate* recursion. This means that we have a single recursive Horn clause, and that the recursive relation appears in the antecedent only once. Such a recursive function-free Horn clause will have the following form:

$$\mathbf{P}(\underline{x}^{(0)}) \wedge \mathbf{Q}_1(\underline{x}^{(1)}) \wedge \cdots \wedge \mathbf{Q}_k(\underline{x}^{(k)}) \rightarrow \mathbf{P}(\underline{x}^{(k+1)}), \tag{1}$$

where for each i, $\underline{x}^{(i)}$ is a subset of some fixed set of variables $\{x_1,x_2,...,x_n\}$, $\mathbf{P}$ is a virtual relation, and $\{\mathbf{Q}_i\}$ is a fixed set of stored relations. As analyzed in [Ioan86c], the problem of recursion can be defined in operator form as follows. A recursive Horn clause, such as the one shown above, may be considered as some relational operator A, applied on some relation $\mathbf{P}$, to produce more tuples for the same relation. So, it can be written as $A\mathbf{P} \subseteq \mathbf{P}$, where A is an operator that maps relations over a fixed set of domains into relations over the same set of domains. The relations in $\{\mathbf{Q}_i\}$ are the parameters of A. If we employ this approach, we are able to define operations on relational operators as follows. *Multiplication* of operators is defined by

$$(A * B)\mathbf{P} = A(B\mathbf{P}),$$

and *addition* is defined by

$$(A+B)\mathbf{P} = A\mathbf{P} \cup B\mathbf{P}.$$

For notational convenience, we omit the operator $*$. Identity ($1\mathbf{P} = \mathbf{P}$), and null ($0\mathbf{P} = \varnothing$, $\varnothing$ the empty set) are defined in obvious ways. The n-th power of an operator A is inductively defined as:

$$A^0 = 1,\ A^n = A * A^{n-1} = A^{n-1} * A$$

The algebraic structure thus obtained is a *closed semiring* [Aho74].

Having established an algebraic framework, the problem of immediate recursion can now be stated as follows: Assume that we have a recursive Horn clause that can be represented by the operator A, so that

$$A\mathbf{P} \subseteq \mathbf{P}.$$

Also, there exists some stored relation $\mathbf{Q}$, which is either stored or produced by some other set of Horn

clauses not involving **P**, so that

$$\mathbf{Q} \subseteq \mathbf{P}.$$

Then, the minimal relation defined by the given set of Horn clauses can be found as the solution to the equation

$$\mathbf{P} = A\,\mathbf{P} \cup \mathbf{Q}. \tag{2}$$

Presumably, the solution is a function of **Q**; we can write $\mathbf{P} = B\,\mathbf{Q}$, and the problem becomes one of finding operator B. Manipulation of (2) results in the elimination of **Q**, so that we have an equation of operators only. In this pure operator form, the recursion problem can be restated as follows: Given some operator A, find another one B satisfying:

(a) $1 + A\,B = B$, and

(b) B is minimal with respect to (a), i.e. for all other C satisfying (a), it is $B \leq C$.

The solution to the above equation (a), under constraint (b), was shown in [Ioan86c] to be equal to

$$A^* = \sum_{k=0}^{\infty} A^k.$$

The operator A^* is called the *transitive closure* of A, which taking into account the definitions of + and $*$, can also be written as

$$A^* = \lim_{k \to \infty} (1 + A)^k. \tag{3}$$

Since A does not contain any functions, for every finite relation **Q** there exists some N (depending on **Q**), such that

$$A^*\,\mathbf{Q} = \sum_{k=0}^{N} A\,\mathbf{Q} = (1 + A)^N\,\mathbf{Q}.$$

This says the fortunate and somewhat obvious fact that, when dealing with finite relations (which is the case in a database environment), only a finite number of the terms of the sum are needed to produce the complete answer. Hence, A^* is an operator that maps finite relations to finite relations.

We mentioned above that the minimal solution of (2) is of the form $\mathbf{P} = A^*\,\mathbf{Q}$. Therefore, any query on relation **P** can be transformed to a query on $A^*\,\mathbf{Q}$. For example, the query $\mathbf{P}(c,...)?$, which asks for the tuples of **P** that have the constant c in the first column p_1, will be the query $\sigma_{p_1 = c}\,A^*\,\mathbf{Q}$, which is a query on base relations. For the remainder of this paper, the stored relation **Q** is ignored. The forthcoming

analysis and description of the various algorithms is in terms of computing A^*.

3. STRATEGY SPACE

Consider a linear operator A and its transitive closure A^*. A *strategy* to compute A^* is a sequence of multiplications and additions of simpler algebraic expressions (operators) forming A^*. Let S be the set of all the equivalent algebraic forms of A^*. Each member s of S has an associated cost $c(s)$. The goal of any optimization algorithm is to find the member s_0 of S, such that

$$c(s_0) = \min_{s \in S} c(s).$$

Whenever the cardinality of S is large, performing an exhaustive search is impossible.

Certain algebraic expressions are members of S independent of the form of A itself. For example, $\sum_{k=0}^{\infty} A$ is always a member of S by definition. Depending on the form of A, however, additional algebraic expressions may be equal to A^*, and hence be members of S. In fact, the larger the number of special properties A has, the larger S becomes. For example, assume that $A = B\,C$ (B, C two operators) and $B\,C = C\,B$. In that case, A^* can be written as $A^* = \sum_{k=0}^{\infty} B^k\,C^k$, which is therefore a legitimate strategy to compute A^*.

An example of such a strategy is the *counting* algorithm that has been proposed by Bancilhon et. al [Banc86c]. and generalized by Sacca and Zaniolo [Sacc86]. In the original proposal this algorithm is applied to a specific Horn clause, namely the same generation cousin example:

$$\mathbf{SG}(w,z) \wedge \mathbf{P}_1(\mathrm{w},\mathrm{x}) \wedge \mathbf{P}_2(\mathrm{z},\mathrm{y}) \rightarrow \mathbf{SG}(\mathrm{x},\mathrm{y}). \tag{4}$$

Basically, counting computes the transitive closure of $\mathbf{P}_1$ and $\mathbf{P}_2$, storing for each tuple produced the level of iteration in which it was generated. It then proceeds to match tuples that were generated in the same iteration for the final result.

The relational operator A applied on **SG** in (4) is a multiplication of two joins, one having $\mathbf{P}_1$ as a parameter and another having $\mathbf{P}_2$ as a parameter (for clarity, we do not specify the join columns and we omit the projections at the end):

$$A = (\mathbf{P}_1 \bowtie)(\mathbf{P}_2 \bowtie). \tag{5}$$

The key observation is that the two joins, $(\mathbf{P}_1 \bowtie)$ and $(\mathbf{P}_2 \bowtie)$, commute with each other. Hence, A^* can

be written as $A^* = \sum_{k=0}^{\infty} B^k C^k$. Counting has been shown to outperform several other algorithms in many cases [Banc86a, Banc86b]. So, it is almost always advantageous to have this strategy in the state space S.

Not all query optimizers will explore the complete solution space. Which part of S is worth exploring is the implementor's decision. The smaller the explored space is, the higher the probability that the optimum is missed, and the faster the optimization algorithm runs.

Notice that in this formulation, we consider strategies that differ only in their algebraic (syntactic) form. For the purposes of this paper, we have ignored several parameters of the optimization problem; i.e., the possibility of the existence of a variety of storage structures through which a relation may be accessed, the variety of algorithms available to execute a join between two relations, and the possibility of executing some operators in parallel. The strategy space is formed simply by the syntactically varying forms of the transitive closure A^* of some operator A. A query optimizer is expected to take these parameters into account as in conventional query optimization.

The question that arises is whether considering all these strategies is of any worth. The presence of a strategy in the space considered is justified only if there is some probability that the cost of the strategy is the global optimum of the cost function defined on the strategy space. Although the relative cost of a strategy is, in general, database dependent, there is a limited number of cases for which the suboptimality of a strategy is provable by purely syntactic means, i.e., independent of the database. Any strategy that can be proved to be suboptimal, (i.e., there is always another strategy with cost no higher than the cost of the first strategy), can be removed from the considered strategy space. Clearly, it is in one's best interest to identify as many such strategies as possible so that the size of the strategy space is minimized.

Bancilhon has used the term *duplicate-free* strategy for strategies such that no two operators are multiplied with each other more than once [Banc85]. He has also proved that duplicate-free strategies are suboptimal. Using this criterion, he has shown that $(1 + A)^N$, which corresponds to the *naive evaluation*, is not duplicate-free and, therefore, not cost effective. Bancilhon's suboptimality criterion is extended as follows:

Definition 3.1: A strategy is *repetition-free* if it forms each algebraic expression exactly once.

Notice that a duplicate-free strategy is repetition-free also. However the reverse is not true. For example, producing both $A\,(A\,A)$ and $(A\,A)A$ in a strategy is not repetition-free even though it is duplicate-free. Clearly, a strategy that is not repetition-free is not cost effective since it includes redundant computation. Identifying non-repetition-free strategies and removing them from the strategy space considered is highly desirable. Theorem 3.1 provides a result in this direction.

Theorem 3.1: Consider two algebraic expressions B and C, such that $B = A^n$ and $C = \sum_{i=1}^{m} A^{k_i}$. Consider a strategy that involves the multiplication $B\,C$. If $n \in \{k_j - k_l : 1 \le j, l \le m\}$ then the strategy is not repetition-free.

Proof: Let $B = A^n$ with $n = k_j - k_l$, for some $1 \le j, l \le m$. Multiplying B with C produces the sum of all the powers of A of the form A^{n+k_i}, $1 \le i \le m$. For $i = l$, operator $A^{n+k_l} = A^{k_j - k_l + k_l} = A^{k_j}$ gets produced, which has already been formed before as part of C. Therefore, multiplying B and C prohibits the strategy from being repetition-free □

Example 3.1: Consider the following formula, which corresponds to the naive evaluation of [Banc85]:

$$A^* = \lim_{k \to \infty} (1 + A)^k = \cdots (1 + A)(1 + A).$$

Its inefficiency follows directly from Theorem 3.1. Naive evaluation is not repetition-free, because A is multiplied with $(1 + A)$, which corresponds to the values $n=1$, $k_j=0$ and $k_l=1$ in the statement of the theorem. □

Although there is a unique algebraic expression equal to A^* associated with each evaluation of A^*, the opposite does not hold. There is certain algorithmic information lost when a strategy is "flattened out" to an algebraic expression. In particular, whether a repeated subexpression is computed only once or not cannot be decided from the algebraic expression alone. For example, consider the expression $A^2 + (A^2)A$. There is no indication of whether A^2 is computed once or twice. For this reason, a strategy is represented by a directed acyclic graph, whose leaves are the primitive operators involved in the computation of A^*, and the other nodes are multiplications and additions applied on their children. In the examples that follow, the edges of the graphs always have top to bottom direction. Notice that a strategy is, in general, a graph and not a tree, since the same expression may appear in multiple places in A^*. Unless otherwise specified,

we assume that each expression is computed only once.

Example 3.2: The graphs in Figure 3.1 represent the states $\sum_{k=0}^{2} A^k$ and $(1+A)^2$ respectively. Also, the algebraic expression $A^2 + (A^2)A$ may correspond to any of the graphs in Figure 3.2, depending on whether A^2 is computed once or twice.

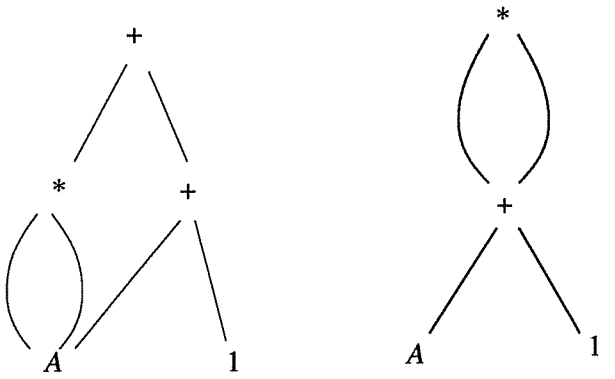

Figure 3.1. Strategies corresponding to $1 + A + A^2$ and $(1+A)^2$.

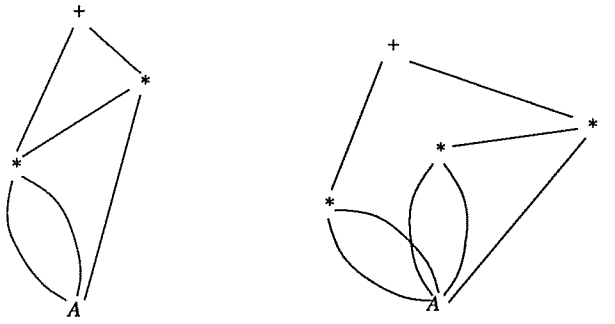

Figure 3.2. Two different strategies corresponding to $A^2 + (A^2)A$. □

Definition 3.2: The *depth* of a strategy is the depth of its corresponding graph, i.e., the maximum path-length in the graph.

Notice that all the graphs corresponding to a single algebraic expression have the same depth. Hence, depth is well defined for an algebraic expression also.

Example 3.3: The algebraic expression $1 + A$ has depth 1, whereas $A\ B + C\ D$ has depth 2 (see Figure 3.3).

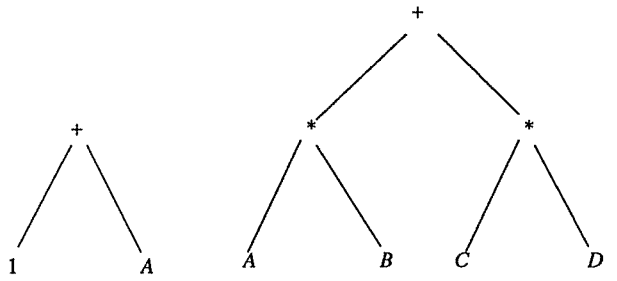

Figure 3.3. $1 + A$ of depth 1 and $A\ B + C\ D$ of depth 2. □

For the remainder of this paper, a strategy is identified with its graph. Occasionally, if this does not cause any confusion, the corresponding algebraic expression is used.

In the following two sections, a simulated annealing algorithm for the optimization of the computation of A^* is described.

4. OPTIMIZATION BY SIMULATED ANNEALING

Simulated annealing is a *Monte Carlo* optimization technique proposed by Kirkpatrick, Gelatt, and Vecchi for complex problems that involve many degrees of freedom [Kirk83]. Such problems are modeled by a state space, where each state corresponds to a solution to the problem. A cost is associated with each state, and the goal is to find the state associated with the globally minimum cost. For complex problems with very large state space, exhaustive exploration of all the states is impractical. Probabilistic hill climbing algorithms, such as simulated annealing, attempt to find the global minimum by traversing only part of the state space. They move from state to state allowing both downhill and uphill moves, i.e., moves that reduce and moves that increase the cost of the state respectively. The purpose of the latter kind of moves is to allow the algorithm to escape from local minima it may occasionally encounter. For example, consider the one dimensional function of Figure 4.1. States S_1 and S_2 are local minima, whereas S_3 is the global minimum. A probabilistic hill climbing algorithm works in such a way that, even if at any point finds itself in state S_2, with high probability it climbs up again to eventually terminate in S_3.

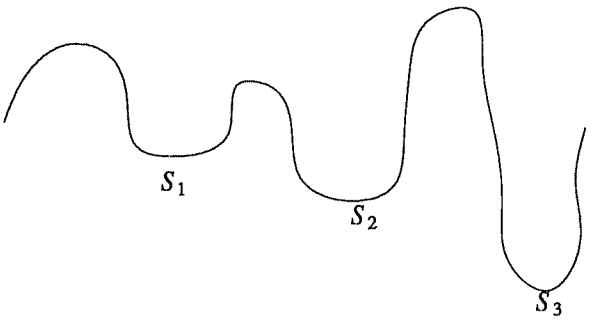

Figure 4.1. Local and global minima.

In simulated annealing the uphill moves are controlled by a parameter T, the temperature. The higher T is the higher the probability an uphill move is taken. As time passes, T decreases, and at the end, when the system is "frozen" ($T = 0$), the probability of making an uphill move is negligible. This algorithm simulates the annealing process of growing a crystal in a fluid by melting the fluid (high T) and then slowly decreasing T until the crystal is formed. At the end of the process, the fluid is at its lowest energy state. In optimizing by simulated annealing, the cost function plays the role of the energy in the physical phenomenon. The analogy between the physical process of annealing and simulated annealing is drawn in Table 4.1.

Annealing	Simulated Annealing
Temperature	Time
Energy	Cost Function
Crystal Formation	State

Table 4.1. Analogy of Annealing and Simulated Annealing.

Simulated annealing works as follows: Consider a state space S_A (A for Annealing), and a function $N_A: S_A \rightarrow Po(S_A)$[†], such that for a state s, $N_A(s)$ is the set of neighbors of s in S_A. Also, consider a cost function $c_A: S_A \rightarrow \mathbb{R}$ that associates a cost with each state in S_A. Visualizing the state space as the set of nodes in a directed graph, $N_A(s)$ represents the edges emanating from the state (node) s. The state space is assumed to be strongly connected, i.e., there exists a path from any node to any other node. Algorithm 4.1 shows the basic structure of simulated annealing.

[†] For a set S_A, its powerset $Po(S_A)$ is defined as follows: $Po(S_A) = \{T : T \subseteq S_A\}$.

$s = s_0$;
$T = T_0$;
while (not_yet_frozen) **do**
    while (not_yet_in_equilibrium) **do**
        s' = random state in $N_A(s)$; (step 1)
        $\Delta c = c_A(s')\text{-}c_A(s)$;
        if ($\Delta c \leq 0$) **then** $s = s'$;
        if ($\Delta c > 0$) **then** $s = s'$ with probability $e^{-\frac{\Delta c}{T}}$; (step 2)
    T = *reduce* (T);
return(s);

Algorithm 4.1. Simulated Annealing.

There are two major loops in Algorithm 4.1. In the inner loop, the temperature T is kept constant as the algorithm explores part of the state space. Downhill moves are always accepted, but uphill moves are accepted with some probability less than 1. After some form of equilibrium is reached, the temperature is reduced (function *reduce*), and the inner loop is entered again. The whole process stops when the freezing point is reached. Each iteration of the outer loop, which is done at a constant temperature, is called a *stage*.

Many theoretical investigations have been performed on the behavior of the simulated annealing algorithm [Rome84, Rome85, Haje85]. It has been shown that, under certain conditions satisfied by the way the next state is "randomly" chosen (step 1 in Algorithm 4.1) and the way the temperature is reduced (step 2 in Algorithm 4.1), as T approaches 0, the algorithm converges to a state s, such that $c_A(s)$ is the global minimum of c_A. Hence, the original optimization goal is achieved.

5. SIMULATED ANNEALING FOR RECURSION

5.1. Algorithm Formulation

Simulated annealing has been applied to a great variety of optimization problems, often with substantial success. The major field of its application seems to be VLSI design, in particular standard cell placement and global routing [Rome84, Sech86a]. VLSI design had been identified as a potential application of simulated annealing as early as its original proposal in [Kirk83]. However, simulated annealing has been applied to other areas also, such as pattern recognition [Ackl85]. In addition, several experimental studies have been conducted with simulated annealing applied to more traditional optimization problems, in particular graph partitioning, graph coloring, number partitioning, and the traveling salesman problem [Arag84].

The successful application of simulated annealing on this great variety of optimization problems, together with its theoretical foundation and its elegant simplicity, has been the primary motivation to devise a simulated annealing algorithm for recursive query optimization. We have used the algebraic model of Section 2 for this purpose; any other model, however, would be equally appropriate. Although the structure of the simulated annealing algorithm is problem-independent, some parameters of the algorithm depend on the particular problem concerned. These are the state space S_A, the neighbor's set for each state s in S_A (given by the function $N_A(s)$), and the cost function c_A. The definitions of these parameters for recursive query optimization are given in this section. In addition, some parameters of the algorithm are implementation-dependent (and somewhat problem-dependent also); these are specified in the next section, which discusses the implementation of the algorithm.

- **State space**

Each strategy that computes A^* is a state in the state space S_A. According to Section 2, a strategy is a graph. Therefore, the state space is a graph whose nodes are graphs. Since, for any specific computation, A^* is a finite operator, only finite sums of the form $\sum_{k=0}^{N} A^k$ are considered. For example, $\sum_{k=0}^{N} A^k$ and $(1+A)^N$ are two distinct states in the state space. As a continuation of the discussion in Section 1 we want to emphasize that estimating N accurately is a very important and difficult problem. It is, however, a problem orthogonal to the development of the optimization algorithm. That is why, for the purposes of this paper, we assume that N is given.

Following on the discussion in Section 3, the state space can be enhanced according to any special properties A has, or it can be made smaller by applying Theorem 3.1 or other similar results. Needless to say, the state space can also be pruned down heuristically by removing strategies that are likely to be suboptimal. Elimination of states, however, has to be done with great caution, because the reduced state space has to remain strongly connected. Otherwise, the optimal state may not be reachable from the initial state.

• **Neighboring states**

For each state $s \in S_A$, the set of its neighbors $N_A(s)$ is determined by the properties of multiplication and addition within the closed semiring of the linear relational operators. In particular, s' is a neighbor of s, i.e., $s' \in N_A(s)$), if s' can be produced by applying one of the following transformations to a *single* node in the graph of s.

Associativity of +: For three operators A, B, and C, $(A + B) + C = A + (B + C)$ (see Figure 5.1).

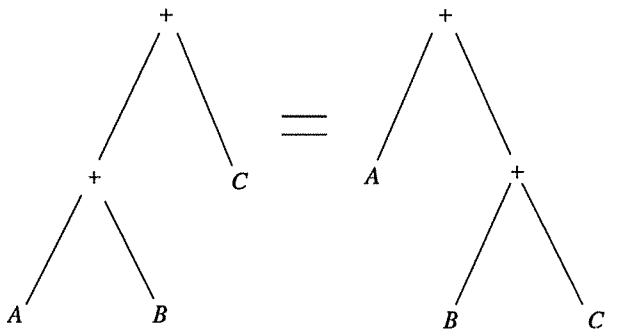

Figure 5.1. State transformation by associativity of +.

Associativity of $*$: For three operators A, B, and C, $(A * B) * C = A * (B * C)$ (see Figure 5.2).

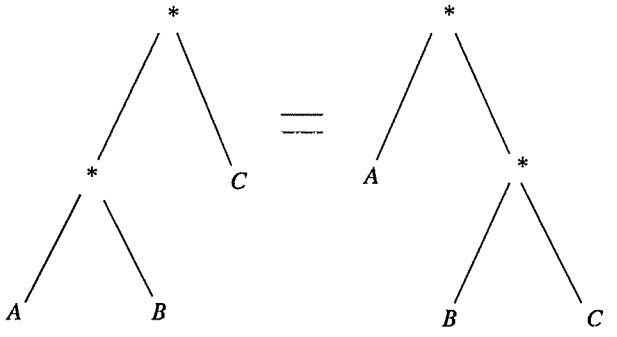

Figure 5.2. State transformation by associativity of $*$.

Commutativity of +: For two operators A and B, $A + B = B + A$ (see Figure 5.3).

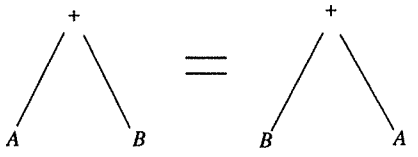

Figure 5.3. State transformation by commutativity of +.

Distributivity of $*$ *over* $+$: For three operators A, B, and C, $A\,(B + C) = A\,B + A\,C$ (see Figure 5.4) and $(B + C)A = B\,A + C\,A$.

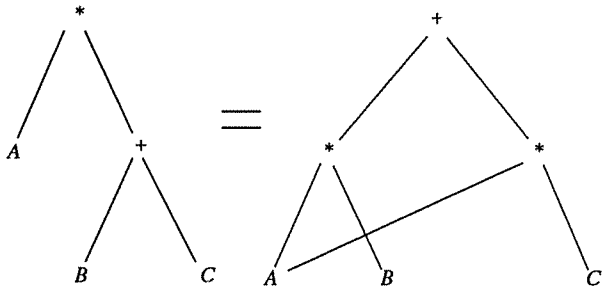

Figure 5.4. State transformation by distributivity of $*$ over $+$.

Distributivity of $*$ *over* $+$ *with 1, the multiplicative identity*: For two operators A and B, $A\,(B + 1) = A\,B + A$ (see Figure 5.5) and $(B + 1)A = B\,A + A$.

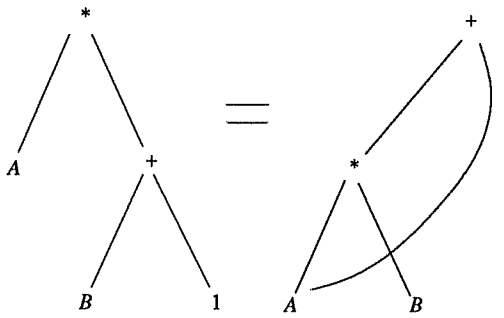

Figure 5.5. State transformation by distributivity of $*$ over $+$, with 1 the multiplicative identity.

Each property in the definition of a closed semiring [Aho74] corresponds to a transformation, as outlined above. There are a few notable exceptions to that. Namely, there are no transformations corresponding to the properties that 0 is the additive identity ($A + 0 = 0 + A = A$), 1 is the multiplicative identity ($A * 1 = 1 * A = A$), 0 is an annihilator ($A * 0 = 0 * A = 0$), and $+$ is idempotent ($A + A = A$). The first three are excluded because they create strategies that are equivalent to strategies already in S_A. The last one is excluded because it creates strategies (states) that are not repetition-free. Its exclusion reduces the state space size by removing states that are known to be suboptimal. For example, the state $(1 + A)^N$ is not considered, since the idempotency of $+$ is not a transformation (e.g. $(1 + A)^2 = 1 + A + A + A^2$).

• **Cost function**

Since the problem addressed is one of database query optimization, the cost function c_A is the cost of applying the individual operators on the operand relations. In a real system, this is an estimate produced with the help of statistics kept by the system about the database.

Modeling the cost c_A of applying relational operators on relations may be done at many levels of detail. In this sense, the choice of the cost function c_A is implementation-dependent also. Producing an accurate model is a difficult problem, even for the case of regular query optimization [Wong76, Seli79, Jark84, Mack86b]. Since it does not affect the applicability or the performance of the simulated annealing algorithm, our description remains general and assumes that c_A is given. In fact, c_A could be an arbitrary cost function, completely unrelated to relational operator costs. The optimization problem would still be well defined by the state space S_A and the neighbors function N_A, and simulated annealing would still be applicable.

5.2. Algorithm Implementation

We have implemented simulated annealing for the optimization of A^* using the state space and neighbor function. The implementation was done in Franz LISP [Wile83], under the Unix 4.3 operating system on a VAX 11/780. LISP was chosen over C, which would be the other obvious choice in our environment, because of the graph form of the states and LISP's ability to manage lists (and therefore graphs) efficiently and elegantly. The cost function c_A is a simple model of the I/O cost of database operations. Join (with projection) and union are the only operations modeled. The cost of a join is the product of the sizes of the two relations plus some additional linear terms to read the relations and write the result. The cost of a union is the sum of the sizes of the two relations to read them plus the size of the result to write it out.

It has already been mentioned that there are some parameters of the simulated annealing algorithm that are implementation-dependent. These are the initial state s_0, the initial temperature T_0, the freezing criterion, the equilibrium criterion, the way the next state is randomly chosen, and the way the temperature is reduced from stage to stage (the *reduce* routine in Algorithm 4.1). In the current implementation they have been chosen as follows:

• **Initial state s_0**

From the theoretical analysis of the simulated annealing algorithm, it has been shown that the effectiveness of the algorithm in finding the global minimum state is independent of the choice of the initial state [Rome84, Rome85, Haje85]. Moreover, this has been verified by many experimental studies with simulating annealing algorithms [Arag84, Sech86a]. Since any initial state is as good as any other, we chose the state that corresponds to the *semi-naive* evaluation [Banc85] as the initial state in our implementation. For example, assuming that only the first two powers of A are to be computed, that is $1 + A + A^2$, the initial state is shown in Figure 5.6.

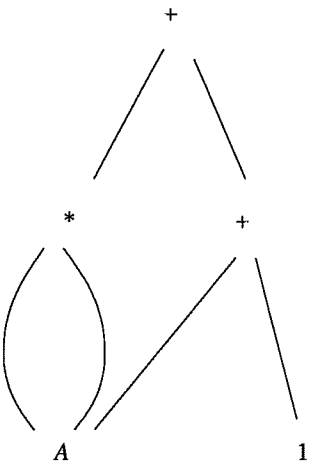

Figure 5.6. Initial state to compute $\sum_{k=0}^{2} A^k$.

• **Initial temperature**

Choosing the appropriate initial temperature T_0 is much more crucial than choosing the initial state. T_0 has to be considerably high so that the system is relatively "hot" in the beginning and allows many uphill moves to be accepted. For the range of experiments performed, the initial temperature was chosen to be twice the cost of the initial state:

$$T_0 = 2\, c_A(s_0).$$

The factor 2 was chosen based on the experience of applying simulated annealing on other domains [Sech86b].

• **Freezing criterion**

There is a great variety of freezing criteria proposed in the literature. Most of them are a combination of tests verifying that the system is at a low temperature and that the system does not change state often, i.e., that it has converged to its final state. In the current implementation, the criterion used is a combination of the ones given in [Arag84] and [Sech86a] and consists of two parts. First, the temperature has to be below 1 ($T \leq 1$); second, the value of c_A at the final state has to be the same for four consecutive stages.

• **Equilibrium criterion**

In all the implementations of simulated annealing that we are aware of, every stage consists of a specific number of iterations through the inner loop. This number may be independent of the temperature of the stage [Arag84], or it may get larger as the temperature decreases [Sech86a]. The advantage of having more state transitions during the later stages of the execution has been theoretically justified as well [Mitr85]. We have chosen to have a constant number of iterations through the inner loop, independent of the temperature. This number is equal to $epoch\_factor * epoch$, where $epoch\_factor$ is an arbitrary factor (chosen to be 16 in our experiments), and $epoch$ is the number of neighbors of the initial solution (both the terminology and formulation are from [Arag84]).

• **Choosing the next transition**

At any point in the execution, the next state is chosen from the set of the current state's neighbors according to some transition probability matrix $R : S_A \times S_A \rightarrow [0,1]$. In general, R must have some specific stochastic properties for the algorithm to converge (it has to be *reversible* [Haje85]). In our implementation, each neighbor of the current state has equal probability to be chosen as the next state. I.e.,

$$R(s,s') = \begin{cases} \dfrac{1}{|N_A(s)|} & \text{if } s' \in N_A(s) \\ 0 & \text{otherwise} \end{cases}.$$

• **Reducing the temperature T**

Many *cooling schedules* have been proposed for the simulated annealing process. We distinguish two of them. Specifically, Hajek [Haje85] proposes reducing the temperature according to the formula

$$T_k = \frac{d}{log(k+1)}.$$

In the above, k is the current stage number (it represents time also), and d is some constant for which he gives a sufficient value for the algorithm to converge to the global minimum. Unfortunately, from a practical point of view, this is not a desirable schedule; it is very slow. For this reason, another schedule has been proposed that reduces the temperature according to the formula

$$T_{new} = \alpha(T_{old})\, T_{old}.$$

The function α takes values between 0 and 1. In [Rome84, Sech86a] α ranges over time (that is, it depends on T_{old}). It is smaller in the beginning (cooling the system fast), then it rises up to higher values (slowing down the cooling process), and eventually it becomes small again to drive the system down to a minimum without any uphill moves. To the contrary, in [Arag84] it is suggested that α should not change, but should remain constant at a relatively high value (in the range of 0.9 to 0.95). We have experimented with both a constant α=0.95 and a variant α modified according to Table 5.1.

$T_0/T \leq$	α
2	0.80
4	0.85
8	0.90
∞	0.95

Table 5.1. Factor to reduce the temperature.

5.3. Experiments with Simulated Annealing

Due to time constraints, we have performed a very small number of experiments with the system. We have used small examples with three and seven terms of A^*, i.e., $\sum_{k=0}^{N} A^k$, N=3 or N=7. In all the experiments, A was the operator corresponding to the **ancestor** Horn clause:

$$\textbf{ancestor}\,(x,z) \wedge \textbf{father}\,(z,y) \rightarrow \textbf{ancestor}\,(x,y).$$

The size of **father** was always one page. This represents an unfavorable situation for simulated annealing for the following reasons:

- The state space is relatively small.

- A has a simple form.
- The size of the relations are small.

We have already mentioned that simulated annealing is inappropriate for small state spaces due to the overhead involved in the execution of the optimization algorithm itself. In addition, the more complex the optimized operator the greater the variety of plans generated, and therefore the larger the solution space is. Finally, with small relations one cannot expect significant variations in the cost of the various states, and this causes problems to simulated annealing (see Section 6).

Table 5.2 shows the results of the experiments. Column c_{semi} gives the cost of the semi-naive evaluation, column c_{low} gives the lowest cost state the algorithm went through, column $c_{\lim}$ gives the cost of the state to which the algorithm converged, and column c_{opt} gives the actual optimum. Also, column Stages gives the total number of stages the algorithm needed to converge.

epoch_scale	N	T_0	T_∞	Stages	c_{semi}	c_{low}	$c_{\lim}$	c_{opt}
16	3	24	0.44	39	12	6	6	6
16	7	72	0.63	65	36	18	21	18
32	7	72	0.63	65	36	21	21	18

Table 5.2. Experimental results.

Although it is premature to draw any general conclusions from such a limited range of experiments, we can say that the results have been encouraging. For the case with N=3, the algorithm always found the minimum cost state, which incidentally is the *smart* algorithm of [Ioan86b]. For the case with N=7, the algorithm did not always find the global minimum. It did, however, converge to a state with a cost that was close to the global minimum and much smaller than the cost of the original state. Further experimentation is definitely needed to verify the general applicability of the algorithm to recursive query optimization.

6. CRITIQUE

There are a few points in the above formulation where there is room for improvement. The first one is that, given a state s, $|N_A(s)|$ is small. That is, the transformations described in Section 5.1 for changing states give a relatively small number of neighbors to each state. This has the implication that the paths between states tend to be long, and the algorithm takes many steps to traverse them. A solution is to allow

the algorithm to perform many transformations at once (as a single move), thereby putting direct transitions among more states than now. We expect that this will reduce the number of stages the algorithm needs to converge significantly. It is not obvious what the optimal number of combined transformations is. This is an issue that we plan to investigate in the future.

The solution proposed to the first problem affects the second one also. That is, the cost change (Δc in Algorithm 4.1) when making a transition is usually small relatively to the total cost. For example, changing the order of two joins (by applying associativity of multiplication) cannot significantly affect the total cost of the whole computation of A^*, if the latter contains many terms. Hence, even at very high temperatures the system does not undergo drastic changes in terms of its cost. It has been experimentally verified that a small range of Δc is most of the time disastrous for the effectiveness of the algorithm [Sech86b]. Combining several transformations into one state transition may give the desired range to Δc and allow the algorithm to perform better.

Finally, the way the next state is chosen in the current implementation also causes problems. Many transformations move between states with the same cost. For example, applying commutativity of + leaves the cost of the state unchanged. This tends to create several *plateaux* in the state space. The system tends to wander in these plateaux for a long time without making progress, going neither downhill nor uphill. Since every applicable transformation is equally likely, the majority of the transitions are of this form, with no change in the cost. The proposed solution is to give higher probability to transitions that do affect the cost than to those that do not. This has to be applied with great caution, however, since paths to efficient states may go through several cost-indifferent transitions before making any significant progress. Nevertheless, this discriminative treatment of neighbors has been used elsewhere with great effectiveness [Sech86a].

7. CONCLUSIONS

Simulated annealing is a probabilistic algorithm, good for optimization problems that involve large solution spaces. Each solution represents a state in the search space, and each state has an associated cost. The goal is to find the state with the minimum cost.

We have adopted simulated annealing to perform query optimization for complex queries that arise in the study of linear recursion. The state space definition has been based on the algebraic structure of relational operators. Each state represents a different algebraic expression for the query answer. The cost of the state is the total I/O cost of the operations in the corresponding algebraic expression.

We have implemented a prototype of the algorithm in LISP and have performed a limited number of small experiments. Our initial experience is that, in general, the algorithm converges to processing strategies that are very close to the optimal. Moreover, the traditional processing strategies, such as the seminaive evaluation, have been found to be, in general, suboptimal.

The scale of the experiments performed is far from complete. We intend to experiment with a larger variety of operators, larger relations, and deeper recursions (i.e., more terms in A^*). We also intend to modify our initial design and implementation of the algorithm according to the points raised in Section 6. The main issues to be examined in this direction are the following:

- Multiple transformations per transition.
- Different transition probabilities from one state to its neighbors.
- Special transformations according to the properties of the operators involved.

8. REFERENCES

[Ackl85]
Ackley, D. H., G. E. Hinton, and T. J. Sejnowski, "A Learning Algorithm for Boltzmann Machines", *Cognitive Science* **9** (1985), pages 147-169.

[Aho74]
Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[Aho79]
Aho, A., Y. Sagiv, and J. Ullman, "Equivalences Among Relational Expressions", *SIAM Computing Journal* **8**, 2 (May 1979), pages 218-246.

[Arag84]
Aragon, C. R., D. S. Johnson, L. A. Megeoch, and C. Schevon, *"Optimization by Simulated Annealing: An Experimental Evaluation"*, unpublished manuscript, October 1984.

[Astr76]
Astrahan, M. et al., "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems* **1**, 2 (June 1976), pages 97-137.

[Banc85]
Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", in *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, February 1985.

[Banc86a]
Bancilhon, F. and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pages 16-52.

[Banc86b]
Bancilhon, F. and R. Ramakrishnan, "Performance Evaluation of Data Intensive Logic Programs", in *Preprints of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, DC, August 1986, pages 284-314.

[Banc86c]
Bancilhon, F., D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", in *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, Boston, MA, March 1986, pages 1-15.

[Bern81]
Bernstein, P. A., N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)", *ACM TODS* **6**, 4 (December 1981), pages 602-625.

[Blas76]
Blasgen, M. W. and K. P. Eswaran, *"On the Evaluation of Queries in a Relational Data Base System"*, Research Report RJ-1745, IBM San Jose, April 1976.

[Codd70]
Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *CACM* **13**, 6 (1970), pages 377-387.

[Epst78]
Epstein, R., M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Data Base System", in *Proceedings of the 1978 ACM-SIGMOD Conference on the Management of Data*, Austin, TX, May 1978, pages 169-180.

[Gall78]
Gallaire, H. and J. Minker, *Logic and Data Bases*, Plenum Press, New York, N.Y., 1978.

[Gall84]
Gallaire, H., J. Minker, and J. M. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys* **16**, 2 (June 1984), pages 153-185.

[Gran81]
Grant, J. and J. Minker, "Optimization in Deductive and Conventional Relational Databases", in *Advances in Data Base Theory, Vol. 1*, Plenum Press, New York, N.Y., 1981, pages 195-234.

[Haje85]
Hajek, B., *"Cooling Schedules for Optimal Annealing"*, unpublished manuscript, January 1985.

[Ioan86a]
Ioannidis, Y. E., "A Time Bound on the Materialization of Some Recursively Defined Views", *Algorithmica* **1**, 4 (October 1986), pages 361-385.

[Ioan86b]
Ioannidis, Y. E., "On the Computation of the Transitive Closure of Relational Operators", *Proc. 12th International VLDB Conference*, Kyoto, Japan, August 1986, pages 403-411.

[Ioan86c]
Ioannidis, Y. E. and E. Wong, "An Algebraic Approach to Recursive Inference", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986, pages 209-223.

[Jark84]
Jarke, M. and J. Koch, "Query Optimization in Database Systems", *ACM Computing Surveys* **16**, 2 (June 1984), pages 111-152.

[Kers86a]
Kerschberg, L., *Expert Database Systems, Proceedings from the First International Workshop*, Benjamin/Cummings, Inc., Menlo Park, CA, 1986.

[Kers86b]
Kerschberg, L., *Proceedings from the First International Conference on Expert Database Systems* , Charleston, SC, April 1986.

[Kim86]
Kim, W., D. Reiner, and D. Batory, *Query Processing in Database Systems*, Springer Verlag, 1986.

[Kirk83]
Kirkpatrick, S., C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science* **220**, 4598 (May 1983), pages 671-680.

[Kooi80]
Kooi, R. P., *"The Optimization of Queries in Relational Databases"*, Ph.D. Thesis, Case Western Reserve University, September 1980.

[Kris86]
Krishnamurthy, R., H. Boral, and C. Zaniolo, "Optimization of Nonrecursive Queries", *Proc. 12th International VLDB Conference*, Kyoto, Japan, August 1986, pages 128-137.

[Mack86a]
Mackert, L. F. and G. M. Lohman, "R* Validation and Performance Evaluation for Distributed Queries", *Proc. 12th International VLDB Conference*, Kyoto, Japan, August 1986, pages 149-159.

[Mack86b]
Mackert, L. F. and G. M. Lohman, "R* Validation and Performance Evaluation for Local Queries", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pages 84-95.

[Mitr85]
Mitra, D., F. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behavior of Simulated Annealing", in *Proc. 24th Conference on Decision and Control*, Ft. Lauderdale, FL, December 1985.

[Naug86a]
Naughton, J., "Data Independent Recursion in Deductive Databases", in *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, Boston, MA, March 1986, pages 267-279.

[Naug86b]
Naughton, J., *"Optimizing Function-Free Recursive Inference Rules"*, Computer Science Technical Report, Stanford University, Stanford, CA, May 1986.

[Rich83]
Rich, E., *Artificial Intelligence*, McGraw-Hill, New York, N.Y., 1983.

[Rome84]
Romeo, F., A. Sangiovanni-Vincentelli, and C. Sechen, "Research on Simulated Annealing at Berkeley", in *Proc. 1984 IEEE International Conference on Computer Design*, Port Chester, N.Y., October 1984, pages 652-657.

[Rome85]
Romeo, F., A. Sangiovanni-Vincentelli, and ""Probabilistic Hill Climbing Algorithms: Properties and Applications", , in *Proc. 1985 Chapel Hill Conference on VLSI*, edited by H. Fuchs, Computer Science Press, Chapel Hill, N.C., 1985, pages 393-417.

[Rose80]
Rosenkrantz, D. J. and H. B. Hunt III, "Processing Conjunctive Predicates and Queries", *Proc. 6th International VLDB Conference*, Montreal, Canada, October 1980, pages 64-72.

[Sacc86]
Sacca, D. and C. Zaniolo, "The Generalized Counting Method for Recursive Logic Queries", in *Proceedings of the International Conference on Database Theory*, Rome, Italy, October 1986.

[Sagi86]
Sagiv, Y., "Optimizing Datalog Programs", in *Preprints of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, DC, August 1986, pages 136-162.

[Sech86a]
Sechen, C. and A. Sangiovanni-Vincentelli, "TimberWolf 3.2: A New Standard Cell Placement and Global Routing Package", in *Proc. Design Automation Conference*, Las Vegas, NV, June 1986.

[Sech86b]
Sechen, C., *private communication*, June 1986.

[Seli79]
Selinger, P. et al., "Access Path Selection in a Relational Data Base System", in *Proceedings of the 1979 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1979, pages 23-34.

[Sell86]
Sellis, T. K., "Global Query Optimization", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pages 191-205.

[Ston76]
Stonebraker, M., E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems* **1**, 3 (September 1976), pages 189-222.

[Vald86]
Valduriez, P. and H. Boral, "Evaluation of Recursive Queries Using Join Indices", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, pages 197-208.

[Wile83]
Wilensky, R., *LISPcraft*, 1983.

[Wong76]
Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing", *ACM Transactions on Database Systems* **1**, 3 (September 1976), pages 223-241.