An Empirical Comparison of Randomized Algorithms for Large Join Query Optimization

Sushil J. Louis and **Yongmian Zhang**
Department of Computer Science
University of Nevada - Reno
Reno, NV 89557
sushil@cs.unr.edu

Abstract

Non-traditional database applications need new query optimization algorithms to speed up large join queries. In the last decade, general techniques such as iterative improvement and simulated annealing, have been extensively investigated for solving large join query optimization problems. In this paper, we compare a genetic algorithm with iterative improvement and simulated annealing for the optimization of large join queries. We compare the performance of these algorithms by testing them on various types of query strategies. In all of our cases, experimental results show that genetic algorithms performed consistently better than simulated annealing and iterative improvement in terms of both output quality and running time. In addition, we found that it is comparatively easier to tune the parameters of genetic algorithms and drive it to a desired optimal solution. We believe that the genetic algorithm approach ranks fairly high among the algorithms we tested, and hence appears to be a promising approach for large join query optimization in future database management systems.

Introduction [1]

Efficient methods of processing queries are a crucial prerequisite for the success of generalized database management systems. The computing complexity of this optimization process is dominated by the number of such possible sequences that must be evaluated by the optimizer. Thus join query optimization problem can be reduced to a combinatorial problem. One of the major decisions an optimizer must make is the order in which to join the tables referenced in the query. It is not easy for the database system to execute queries quickly because the numbers of alternative strategies to answer a query grows exponentially with the number of relations participating in the query. Thus we need an optimization phase to select the most efficient implementation among the many possible ways to implement a given query.

Most of the existing work on query optimization stems from heuristic elimination methods to reduce the size of the search space of problem sequences. Current query optimization algorithms do well when a relatively small number (such as 10 joins) of relations are involved resulting in a search space of manageable size. System R algorithms (Selinger, Astrahan, & Chamberlin 1979) are an example. They performs optimization by exhaustively searching the problem space with time complexity of $O(2^N)$, where N is the number of relations to join. When join size grows modestly large this algorithm becomes intractable.

Krishnamurthy, Boral, and Zaniolo proposed a kind of heuristic approach to handle queries with a large number of joins (Krishnamurthy, Boral, & Zaniolo 1986). The time complexity of their algorithm is $O(N^2)$, where N is number of query joins. The limitation of this approach is that it depends on a particular form of cost function.

Randomized algorithms for large join queries have been extensively investigated by taking advantage of methods used in various combinatorial optimization problems. *Simulated Annealing* (SA) (Ioannidis & Wong 1987; Swami & Gupta 1988; Ioannidis & Kang 1990) and *Iterative Improvement* (II) (Ioannidis & Kang 1990; Swami & Gupta 1988) were introduced to the area of database query optimization in 1987. Interestingly enough, the experiments carried out by both (Ioannidis & Kang 1990) and (Swami & Gupta 1988) led to opposite results. In (Swami & Gupta 1988), Swami and Gupta concluded that the simple II algorithm was better than SA, but in (Ioannidis & Kang 1990), Ioannidis and Kang indicated that the overhead costs of SA are lower than that of II. Our naive analysis is that, since the overhead costs generated by one algorithm are not significantly lower than that of another when they are applied to large queries, some difference in their experimental criteria, such as annealing schedule, may have led to the differing conclusions. A few hybrid approaches motivated by the above algorithms have also been developed to overcome the deficiencies of each algorithm and improve output quality. These algorithms, such as Two-Phase Optimization (Ioannidis & Kang 1990) which is a combination of SA and II, and SAK (Swami 1989) which uses the KBZ heuristic to generate an initial state for SA, show limited improve-

ment.

Recently, genetic algorithms (GAs) have been widely applied to effectively solve difficult optimization problems (DeJong 1980; Goldberg 1989; Louis 1993). These algorithms have proven to be efficient for complex problems which are not computationally tractable using other approaches. (Bennett, Ferris, & Ioannidis 1993) presented encouraging results by applying genetic algorithms to query optimization problems. Their results for small queries show that the output quality and the running time is, in general, better than the current System-R algorithms if appropriate query representation and GA parameters are chosen. However, the System-R query optimizer will become infeasible as query size increase beyond 10. Based on our literature survey, genetic algorithms have not been explored for large join query optimization so far.

In our study, we investigate the performance of GA, SA, and II on large join queries. We experimentally compare results using these randomized algorithms and determine which techniques are the most effective for large join query optimization. The problems that arise in performing such a comparison are also discussed in this paper.

The rest of this paper is organized as follows. In section 2 we give a framework for query optimization and describe the evaluation function and query strategy space. In section 3, we provide a generic description of the randomized algorithms we used in this paper. Section 4 describes how randomized algorithms can be adapted to query optimization. Then, in section 5, we discuss how we determine the parameters of all these algorithms. We present comparison results and analysis in section 6. Conclusions and direction for future studies are presented in section 7.

The Query Problem

There are two basic kinds of optimization – algebraic manipulation and cost estimation strategies -- found in query processors. Algebraic manipulation is intended to improve the cost of answering the query by means of simplifying queries without consideration of the actual data and its structure, while the cost estimation strategies select a query plan based on the data and data structure of the current database. The randomized algorithm approaches proposed recently for query optimization are oriented toward the cost estimation of different strategies. We assume that the query strategy only has natural joins or equijoins. Although the cost model described in the following section is derived based on natural joins of two relations, if the query consists of equijoins, we can use the same techniques as that used in natural join if we pretend that attributes of one relation are renamed.

Evaluating Join Sequences

A query optimizer in a relational DBMS translates nonprocedural queries into a procedural plan for execution typically by generating many alternative plans, estimating the execution cost of each, and choosing the plan having the lowest estimated cost where cost is generated by

$$Cost(Q(s_0)) = MIN_{s \in S}[Cost(Q(S))] \qquad (1)$$

S is the set of all alternative plans generated by the optimizer and s_0 is a member of S. This can be formally stated as follows: Given a query Q with execution space S, find an execution in s_0 that is of minimum output cost. The cost for merge scan join method can be generally stated as:

$$Cost(R \bowtie S) = 2\,B_R\,log_M B_S + 2\,B_S\,log_M B_S + B_R + B_S + (B_R\,T_S + T_R\,B_S)/I \qquad (2)$$

and for nested loop join method can be expressed as:

$$Cost(R \bowtie S) = 2\,B_R\,(T_S/I + B_S/(M-1)) + B_S\,(T_R/I + 1) \qquad (3)$$

where R and S are two relations, T_R and T_S are the number of tuples in relation R and S, B is the number of blocks in which all the tuples of that relation could fit, M is the number of blocks that can fit in main memory at any one time and I is image size (Ullman 1988). Since the join operation is implemented in most DBMS as a two way operator, the optimizer must generate plans that achieve N way joins as sequences of two joins as described below.

Join Methods

We restrict our work to queries involving only joins in our study. Thus the problem is reduced to deciding on the best join order, together with the best join methods to be used for each join operation. The size of search spaces depends not only on the number of relations to be joined, but also on the number of join methods supported by the system and the index structures that exist on the each relation. We use two join methods in our study:

1. nested loop(NL) in which two relations are scanned in a nested fashion to find tuples in the cross product that satisfy the join predicate.
2. merge scan (MS) in which two relations are sorted on join column values, and the sorted relations are merged to obtain the result.

Randomized Algorithms

Iterative Improvement

The iterative improvement algorithm (II) was initially used to attack the Traveling Salesman Problem (TSP) (Lin & Kernighan 1973). II starts with a random feasible solution. The solution is improved in the objective function by repeatedly generating a random move, based on certain transformation rules, and accepting the move if it lowers the objective function value. This process is repeated until no further improvement can

be found or an appropriate stopping criterion is satisfied. II does not accept any solution of the objective function that is worse than the current solution, that is, it always runs forward.

Simulated Annealing

Simulated Annealing (SA) is motivated by an analogy with the statistical mechanics of annealing of solids (Kirkpatrick, Gelatt, & Vecchi 1983). If a physical system is at a high temperature, then the large number of atoms in the system is in a highly disordered state. To get the atoms into a more orderly state, we need to reduce the energy of the system by lowering the temperature. The system will be in thermal equilibrium when the probability of a certain state is governed by a Boltzmann distribution:

$$Pr(\Delta E) \sim exp((-\Delta E)/kT) \qquad (4)$$

A candidate configuration is generated by randomly perturbing the current configuration and its objective function value is calculated. If the objective function value is lower than the current value, then the displacement is accepted. Otherwise this new displacement is accepted with a probability given by the Boltzmann distribution in Equation 4. Thus there is always a nonzero probability of accepting worse solutions. This gives the algorithm a probability of escaping a local minimum and leads to a global optimum if annealing proceed slowly enough.

Genetic Algorithm

Genetic Algorithms (GAs) are stochastic, parallel search algorithms based on the mechanics of natural selection and the process of evolution (Holland 1975; Goldberg 1989). GAs were designed to efficiently search large, non-linear spaces where expert knowledge is lacking or difficult to encode and where traditional optimization technique fail. GAs perform a multi-directional search by maintaining a population of potential solutions usually encoded as bit strings and encourage information formation and exchange between these solutions. A population is modified by the probabilistic application of the genetic operators from one generation to the next. Whenever some individuals in the population exhibit better than average performance, the genetic features of these individuals will be copied to the next generation.

Conceptually, GAs work with a rich population and simultaneously climb many peaks in parallel during search processing. This significantly reduces the probability of getting trapped at a local minimum. They are thought to be more robust and more broadly applicable than other similar search algorithms.

Application to Query Optimization

We now consider the order in which the relations are to be joined. Although the cardinality of the join of N relations is the same regardless of join order, the cost of joining in different order can be substantially different. First of all, we assume that unnecessary Cartesian products have been excluded in our query strategies and that the system supports the merge scan and nested loop join methods. Thus the query optimization problem can be viewed as finding the best way to join subsets of the relations in a given query plan. Each strategy represents a choice of access paths for the corresponding join processing tree and join methods that the system can support. A query strategy is conveniently represented in the form:

$$S = \begin{cases} K^J & : \text{ if PT is linear tree} \\ K^J_O & : \text{ if PT is bushy tree} \end{cases}$$

$$\text{here } J \in \{NL, MS\} \text{ and } O \in \{a, r\}$$

where S is the query state space; K is a string configuration which consists of sets of natural number to represent relations if we use the linear processing tree, or number of joins if we use the bushy processing tree; J is join method, which is either merge scan (MS) or nested loop (NL); and O is the orientation of their constituent relation. Here a means joined relations in one order while r represents joined relations in reverse order. There are two major differences between the representations of linear processing tree and bushy processing tree (Bennett, Ferris, & Ioannidis 1993). First, the Cartesian products can be eliminated in the bushy representation. Second, because the bushy representation is based on labeling joins, there may be more than one intermediate relation in use at any given processing step. From genetic algorithms point of view, we need a set of query sequences in the form of a string.

Candidate State in SA and II

Candidate states are randomly generated based on certain transformation rules from state to state. The transformation rules used in simulated annealing and iterative improvement are much simpler than crossover and mutation in genetic algorithms. Let S be a candidate space, and let $S1$ and $S2$ be candidate spaces generated by different transformation strategies. The transformation strategy can be compactly described as follows:

$$S \in \{S1, S2\}$$

$$\text{s.t. } Pr\{S1\} = 0.5 \text{ and } Pr\{S2\} = 0.5$$

where Pr is probability distribution. $S1$ and $S2$ are generated based on the following transformation rules.

1. S1: Randomly generate two join nodes. If interchanging these two nodes results in a valid permutation, interchange them and leave others unchanged. The new state will be evaluated based on the cost model. The example is

$$S1 : (\mathbf{5}^M 3^N 4^N 1^M \mathbf{2}^M 6^N 7^N 8^M 9^N) \Longrightarrow$$
$$(\mathbf{2}^M 3^N 4^N 1^M \mathbf{5}^M 6^N 7^N 8^M 9^N)$$

2. S2: Randomly choose two chunks of the current state with length of two. Then we alter S2 by exchanging the two chunks, provided that the transformed configuration satisfies all constrains and leaves other nodes unchanged. The example is

$$S2 : (\mathbf{5}^M\mathbf{3}^N4^N1^M\mathbf{2}^M\mathbf{6}^N7^N8^M9^N) \Longrightarrow (\mathbf{2}^M\mathbf{6}^N4^N1^M\mathbf{5}^M\mathbf{3}^N7^N8^M9^N)$$

We use either of two join methods $\{M, N\}$ with equal probability. If the bushy tree representation is used, we also set the orientation of two relations $\{a, r\}$ with equal probability. Note that changing the join method does not affect the structure of tree query but represents different query sequence.

Crossover and Mutation in GA

The functionality of crossover and mutation is similar to the transformation used in SA and II. The only difference is that crossover in GAs takes into account two query strategies based on the mechanisms of natural selection, whereas SA and II algorithms just simply operate on one query strategy. The crossover operator can be formally stated as: Given two individual α and β, the crossover operator generates a new feasible individual γ, the descendant of α and β. We tested three kinds of crossover in our work, namely Modified Two Swap(M2S), Chunking and Partially Mapped (PM). These crossovers have also proven to be effective for query optimization and other combinatorial optimization problems elsewhere (Bennett, Ferris, & Ioannidis 1993; Goldberg 1989). We found simple M2S crossover does better whether we use it on the bushy tree query or a linear tree query. We therefore address only the M2S crossover in this study.

Mutation was implemented to produce spontaneous random changes of the join method or the orientation if we use the strategy of a bushy processing tree. It reorders the location on the chromosome.

Testbed

In our experiments, the query size ranged from 20 to 100 relations. We connect each relation into join graph based on a processing tree. The relation cardinalities, selections, and the number of unique values in the join columns was chosen randomly within certain ranges. For the sake of comparison, most numerical features of individual relations in our study are adopted from the previous study in (Swami & Gupta 1988).

Determination of Parameters

Our initial goal was to find the most appropriate parameters for our algorithms. We use a typical query configuration of 40 joins with a linear processing tree to determine parameters. We believe the parameters determined by running this configuration also fit others. The cost is averaged over 10 runs in which the random seed for the first run to the 10^{th} run is linearly increased from 0.1 to 1.0. For each chosen combination of parameters, the average objective value of ten runs is converted into a scaled value. The effect of different parameter value is investigated in order to find a set which can provide better performance than other sets in our study. For GAs, we mainly decide crossover rate, mutation rate, population size and number of iterations. For SA, we need to determine the temperature schedule.

Parameters in SA and II

In II, if a generated state has lower cost compared to the current minimum then we call this the new local minimum and start the outer loop again. We run the GA and II for the same number of evaluations; this is the population size of the GA multiplied by the number of generations. The efficiency of SA depends strongly on the temperature schedule of annealing. A temperature step that is too small will be inefficient in exploring the space because almost all candidate solution will be accepted. On the other hand, if the temperature step is too large, we may lose interesting candidate solutions and get traped in local minima. We use a simple schedule for reducing temperature T:

$$T_i = \left[(X_T)^i \times T_0\right]_{i=0}^{i=N} \qquad (6)$$

Where T_i is i^{th} temperature step; X_T is temperature reduction factor, $0 < X_T < 1$; T_0 is the initial temperature. N is iterations until the temperature falls below 1 ($T_i < 1$). Based on our experiments by trial and error, when T_0 is 1.5× (cost of initial state), X_T is 0.90 and equilibrium equals 10× (numbers of joins), the algorithm produced relatively better performance. The SA stops when the temperature falls below one or the number of evaluations equals that performed by the GA.

Parameters for GAs

The size of the population is an important choice in genetic algorithms. If the population size is too small, the genetic algorithm may converge too quickly; if it is too large, the GA may waste computational resources. The most effective population size is dependent on the problem being solved. We temporarily set crossover rate and mutation rate as 0.8 and 0.005 respectively (standard values). Our experiments with population sizes from 20 to 200 indicate that

the average performance improves dramatically as the population size goes from 20 to 50. Only small improvements result as the population size increases beyond 50. On the other hand, the run time increases linearly with population size. Thus we choose a population size of 50. In terms of time, we get acceptable performance within an acceptable computational time when we run for 300 generations. Thus we fix the number of generations at 300 for all our experiments. After a number of experiments with varying crossover and mutation rates we found that we a crossover rate of 0.95 and mutation rate of 0.05 results in slightly better performance than other combinations.

Experimental Comparison

We generated 5 random queries for 20 joins. We then used these queries as templates within which to generate 5 queries for 30 joins and so on until we had 5 queries for 40, 50, 60, ..., 100 joins. The output cost for each query is the average in minimum cost achieved over 10 runs with different random seeds. For SA, we adjust the initial temperature to ensure that it runs for the same number of evaluations as the GA.

In our experiments, we include linear and bushy processing trees. The GA results show a bushy tree representation is slightly better in output cost than the linear tree representation only when query size becomes relatively large. When both representations are used in II and SA, the difference in output cost is insignificant. But overall these three algorithms are the same whether we use a bushy tree representation or a linear tree representation. Since the linear tree representation is widely used in query optimization and other previous works, we focus on the linear processing tree as the query strategy in the following discussion.

Figure 1 presents typical performance of the three algorithms as a function of query size on one of the five queries. The results on the other four sets of queries are similar. The Y-axis in Figure 1 represents scaled cost, i.e., the ratio of the strategy cost over the minimum cost found by GAs. As the plot shows, the cost from GAs is substantially lower than the other two algorithms through all query sizes. When query size increases, the difference becomes even more significant. It appears that GAs performs better than SA or II in large query optimization problems. Our experiments indicate that there are no significant differences in overhead cost between SA and II.

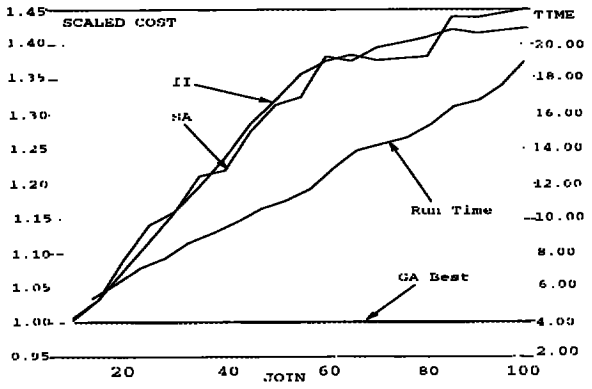

Figure 1: The performance as function of relations

The result given in Figure 2 demonstrates the cost found by the three algorithms as a function of time for a query with size 60. Once again, the scaled cost on the Y-axis is the ratio of the strategy cost over the minimum cost found by GAs, and the X-axis represents the given processing time. From Figure 2, we can see that both SA and II immediately find a good region of the search space, however, this leads to a local minimum. Even with more processing time, they show only slight progress and substantial further improvement is unlikely. Compared against the result of SA and II, in every case, GAs perform better. The more running time, the better solution we can find with GA. This is a practical advantage of our approach for query optimization since a good query strategy is the key issue in speeding up query processing.

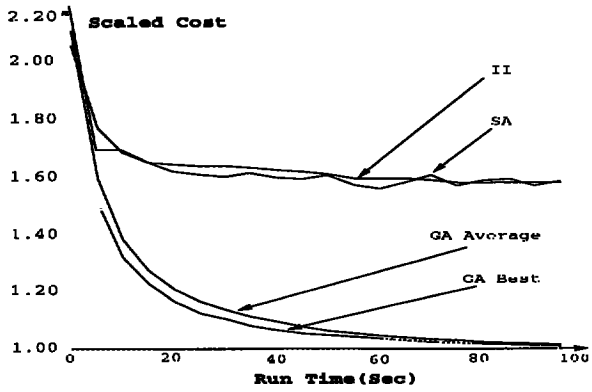

Figure 2: The performance as function of time

We also measured scaled cost of all ten runs for query sizes from 20 to 100 as shown in Figure 3. Scaled cost here is a ratio of strategy cost over best cost of ten runs for each algorithm. As we can see, the overhead scaled cost in GAs is closer to 1 than in the other two algorithms. GAs therefore outperform the other algorithms in terms of stability and reliability. This result is particularly apparent when the query size is less than 60, as can be seen by the profile in Figure 3. We observed

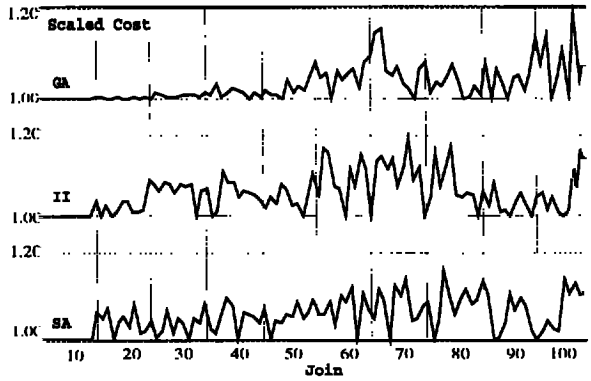

Figure 3: The stability of algorithms

that, in both SA and II, although there is no significant performance difference, II rarely outperforms SA. In our experiments, SA does not show the strong merit described in (Ioannidis & Kang 1990). By searching richer populations based on natural selection, GAs seem have the ability to find a desired global minimum which is hidden among many, poorer, local extrema.

Conclusion

In this paper, we have compared three different randomized algorithms for large query optimization problems and produced experimental results. Theoretically, GAs have extremely attractive features that are rather unique when compared with other randomized algorithms. Our preliminary experimental results show that GAs performed consistently better than SA and II in terms of both output quality and running time. GAs also seem more stable. In addition, we found that it is comparatively easier to tune the parameters of a GA. In general, GAs perform better among the algorithms we tested, and hence appear to be a promising approach for large join query optimization in future database management systems.

In the future, our work will focus on the adaptability of GAs in other database environments, such as distributed database systems. We also intend to use parallel GAs to improve query optimizing speed and make GAs practically applicable for query optimization.

Acknowledgements

This material is partially based upon work supported by the National Science Foundation under Grant No. 9624130.

References

Bennett, K.; Ferris, M. C.; and Ioannidis, Y. E. 1993. Genetic algorithm for database query optimization. In *Proc. of the 5th International Conference on Genetic Algorithms*, 400–407.

DeJong, K. A. 1980. Genetic algorithms: A 10 year perspective. In *Proc. of the First International Conference on Genetic Algorithms*, 169–177.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley.

Holland, J. 1975. *Adaptation In Natural and Artificial Systems.* Ann Arbour: The University of Michigan Press.

Ioannidis, Y. E., and Kang, Y. 1990. Randomized algorithms for optimizing large join queries. In *In Proc. of the 1990 ACM-SIGMOD Conference on the Management of Data*, 312–321.

Ioannidis, Y. E., and Wong, E. 1987. Query optimization by simulated annealing. In *In Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, 9 22.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science* 220(4598):671 680.

Krishnamurthy, R.; Boral, H.; and Zaniolo, C. 1986. Optimization of nonrecursive queries. In *Proc. of the 12th International Conference on Very Large Databases*, 128 137.

Lin, S., and Kernighan, B. W. 1973. An efficient heuristic algorithm for the traveling salesman problem. *Operation Research* 21(2):498–516.

Louis, S. J. 1993. *Genetic Algorithms as a Viable Computational Tool for Design.* Ph.D. Dissertation, Indiana University.

Selinger, P. G.; Astrahan, M. M.; and Chamberlin, D. D. 1979. Access path selection in a relational data base system. In *Proc. of the 1979 ACM-SIGMOD Conference on the Management of data*, 23 34.

Swami, A., and Gupta, A. 1988. Optimization of large join queries. In *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, 8 17.

Swami, A. 1989. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proc. of the 1989 ACM-SIGMOD Conference on the Management of data*, 367–376.

Ullman, J. D. 1988. *Principles of Database Systems.* Rockville, MD: Computer Science Press.