Optimization of Large Join Queries

Arun Swami
Anoop Gupta

Computer Science Department, Stanford University, Stanford, CA 94305

Abstract

We investigate the problem of optimizing Select–Project–Join queries with large numbers of joins. Taking advantage of commonly used heuristics, the problem is reduced to that of determining the optimal join order. This is a hard combinatorial optimization problem. Some general techniques, such as iterative improvement and simulated annealing, have often proved effective in attacking a wide variety of combinatorial optimization problems. In this paper, we apply these general algorithms to the large join query optimization problem. We use the statistical techniques of factorial experiments and analysis of variance (ANOVA) to obtain reliable values for the parameters of these algorithms and to compare these algorithms. One interesting result of our experiments is that the relatively simple iterative improvement proves to be better than all the other algorithms (included the more complex simulated annealing). We also find that the general algorithms do quite well at the maximum time limit.

1 Introduction

The problem of query optimization in relational database systems has received a lot of attention ([JK84] provides a good overview). However, current query optimizers expect to process queries involving only a small number of joins (less than 10 joins). We expect that novel applications built on top of relational systems will require processing of queries with a much larger number of joins. Knowledge base systems using relational systems for storage of persistent data are examples of possible applications. Object–oriented database systems, for instance, Iris [FBC*87], which use relational systems for storage of information are another class of potential applications generating many joins. Krishnamurthy and Boral in [KBZ86] mention applications from logic programming resulting in " ... expressions (similar to database queries) with hundreds (if not thousands) of joins."

In current query optimization algorithms, algebraic transformations and heuristics (such as pushing selections down as much as possible) have been used. These will continue to prove useful. However, strategies for searching large solution spaces are not as well developed. For example, the System/R query optimizer [SAC*79] uses a dynamic programming algorithm whose worst case time complexity is $O(2^N)$ (with an exponential space requirement), where N is the number of joins. Use of this algorithm becomes infeasible as N increases beyond 10.

In [IW87], Ioannidis and Wong study how simulated annealing [KGV83] can be used to obtain an efficient algebraic structure for a given *recursive* query. This too is a problem which involves searching a large solution space. However, they do not consider the problem of optimizing non–recursive queries with a large number of joins.

For optimizing non–recursive queries, an $O(N^2)$ heuristic search algorithm has been described in [KBZ86]. The theory on which their work is based requires that the cost functions have a certain form. To be cast in this form, the cost functions have to be oversimplified. Also, even in the simplified form, all join methods do not have a cost function of the required form. Our work does not depend on any particular cost model; any reasonable cost model will do. For this particular set of experiments we use a cost model for join processing in main memory databases [Swa87a].

In this paper we investigate the problem of optimizing non–recursive large join queries. As will be shown later, the large join query optimization problem (to be denoted by LJQOP) is a hard combinatorial optimization problem. Various general techniques i.e., techniques which are applicable to a wide variety of problems, have been developed for tackling combinatorial optimization problems. In this paper we adapt these techniques to LJQOP and compare them to determine which techniques are the most effective. We also discuss the various problems that arise in performing such a comparison. We plan to investigate heuristic approaches in future work. An example of such a heuristic is the one described in [KBZ86].

The optimization techniques investigated in this paper are general algorithms like simulated annealing and the techniques based on local search [PS82]. In particular, it is of interest to see how useful simulated annealing proves

to be, since experience with this technique has not been uniformly positive [NSS86], [JAMS87]. Besides simulated annealing, techniques based on local optimization, for example, iterative improvement, have also been successfully employed in combinatorial optimization problems. Again though, none of these techniques have been explored in the context of optimizing large join queries.

When comparing different optimization techniques, it is necessary to have a definite criterion to evaluate them. The criterion we use in our study is based on the general principle that we consider a query optimizer to be good in practice if it performs well on the average, and very rarely performs poorly. We will discuss how we translate this principle into an appropriate quantitative measure of the effectiveness of a technique. We also discuss our use of comprehensive statistical methods for both tuning the techniques (obtaining reliable values for the parameters of the techniques) and for comparing the techniques.

The organization of this paper is as follows. In Section 2, we describe the problem of large join query optimization. Assumptions and commonly used simple heuristics are discussed. In Section 3, we describe the different general algorithms which are compared in this experimental work. In Section 4, we describe how these algorithms have been adapted to the specific problem of query optimization. We also describe how test queries are generated. In Section 5, we briefly describe the statistical methods used and present the results of our experiments in tuning and comparing the different algorithms. Finally, in Section 6, we draw some conclusions and describe future research directions.

2 Problem Formulation

Consider a relational algebra query with N joins (by a join, we mean a "two-way" join i.e., a join between two relations) linking $(N+1)$ operand relations (some of these operand relations may represent the same *base* relations). In traditional database applications, N is typically between 0 and 9. A *large join query* is a query where $N \geq 10$. The *large join query optimization problem* (LJQOP) is to find the query evaluation plan (QEP) having the lowest cost. In this paper, we experiment with queries where N is an order of magnitude larger than usual in queries in traditional applications i.e., we consider queries where $10 \leq N \leq 100$.

As is common in query optimization research, we restrict our work to queries involving only selections, projections, and joins (Select–Project–Join queries). We use the simple heuristics of pushing selections down as much as possible, and performing projections as soon as possible. These heuristics do not alter the combinatorial nature of the search space. The major remaining problems in query optimization are deciding on the order in which we join the relations and the join method to be used for each join operation.

In our current experiments, we decided to use only the hash join method. Simulations based on the cost model in [Swa87a] and other proposed models for query processing using large main memory [DKO*84] show that hash–based methods perform well over large ranges of values of the parameters. Also, we need to estimate lower bounds before carry out the optimization as shown in Section 4.2. It is a hard problem to obtain good estimates of lower bounds when other join methods are included. In future work, we will consider how to incorporate the use of multiple join methods in the optimization algorithms. Now, our optimization problem becomes one of choosing the best join order. A QEP can be concisely represented by a join processing tree, abbreviated as JT, and we wish to find a JT having the lowest cost. Note that there may be many JTs with the same cost.

Binary join processing trees (BJTs) are JTs in which each join operator has exactly two operands. A special class of BJTs is the class of *linear join processing trees* (LJTs). In LJTs, of the two join operands, at most one can be an intermediate relation. Most join methods distinguish between the two operands, one being the "outer" relation and the other the "inner" relation. An *outer linear join processing tree* (OLJT) is a LJT in which the inner relation is always a base relation, never an intermediate relation.

The number of different OLJTs is $(N+1)!$, whereas the number of different BJTs is $\binom{2N}{N}N!$. In our experiments, we generate only OLJTs. Most query optimizers, including System R, use the same restriction to cut down on the search space. This is done based on the assumption that a significant portion of the JTs with low processing cost is to be found in the space of OLJTs. The validation of this assumption is an open problem. Another advantage of using LJTs is that they provide increased opportunities for pipelining. Finally, OLJTs, unlike LJTs where the intermediate relations are the inner relations, permit one to take advantage of indexes on join columns of base relations. This is particularly important in nested join methods which prefer join indexes on the inner relations.

Query optimizers restrict the space further by postponing cross–products as late as possible (the intuition is that cross–products are expensive and result in large intermediate results). However, this does not change the combinatorial nature of the search space. In fact, in [IK84], it has been shown that a special case of LJQOP is an NP–complete problem. This shows that the large join query optimization problem is a hard combinatorial optimization problem.

The query optimizer is given a *join graph* representing the join predicates linking the different relations in the query. If the query cannot be evaluated without using a cross–product, the join graph will have at least two components. Using the heuristic of postponing cross–products as late as possible, all such components are processed separately and then the results are joined using cross–products. This means we need concern ourselves only with the optimization of a single component. It is clear that there exists at least one JT for processing a component which does not require a cross–product. Such a JT is called a *valid* JT. We restrict our search to the space of all valid JTs.

In the next section we introduce some terminology from

combinatorial optimization, and then describe different algorithms for tackling such problems. These algorithms, adapted to our query optimization problem, will be compared in Section 5.

3 Combinatorial Optimization Techniques

Each solution to a combinatorial optimization problem can be looked upon as a *state* in a *state space* (in our case the JTs are the states). Each state has a *cost* associated with it as given by some *cost function.* A *move* is a perturbation applied to a solution to get another solution; one *moves* from the state represented by the former solution to the state represented by the latter solution. (We will describe the moves we use in Section 4.1.) A *move set* is the set of moves available to go from one state to another. Any one move is chosen from this move set at random. The probability associated with selecting any particular move is specified along with the move set.

Two states are said to be *adjacent* states (or *neighbouring* states) if one move suffices to go from one state to the other. A *local minimum* in the state space is a state such that its cost is lower than that of all neighbouring states. There can be many such local minima. A *global minimum* is a state which has the lowest cost among all the local minima. There can be more than one global minimum. The optimal solution is a global minimum. A move which takes one to a state with a lower cost is called a *downward* move; otherwise, it is called an *upward* move. In a local or global minimum, all moves are upward moves.

Using this terminology, we now describe the combinatorial optimization techniques that we will compare in Section 5. In all the techniques described below, selection among adjacent states is done at random. Also, the *initial* state is the same for all algorithms and it may be given or it may be generated at random. The initial state must be distinguished from the *start* state which is the state in which a new local optimization run begins, as discussed below.

Perturbation Walk (**PW**) This is the simplest technique. The starting state is a randomly generated state. One keeps moving from the current state to an adjacent state, remembering the lowest cost state visited. The moves are chosen at random. When the algorithm terminates (the stopping criterion could be a time limit; another stopping criterion is discussed in Section 4.2), the lowest cost state encountered is output as the best solution found.

Quasi-random Sampling (**QS**) This technique is somewhat similar to **PW**. Instead of moving to an adjacent state, one generates a new "random" solution. Again, when the algorithm terminates, the lowest cost solution generated is output as the best solution found. The reason for the qualification "quasi" is that we do not generate truly random solutions. Random sampling in the space of *all* JTs is simple (one generates random permutations). However, *truly* random sampling in the space of *valid* JTs is a hard open problem.

```
/* Get an initial solution */
S = initialize();
/* Current minimum cost solution */
minS = S;

repeat {
   repeat {
      /* Randomly selected adjacent state */
      newS = move(S);
      if (cost(newS) < cost(S))
        S = newS;
   } until ("local minimum reached");

   if (cost(S) < cost(minS))
      minS = S;
   /* Obtain a new starting state */
   newStart(S);
} until ("stopping condition satisfied");

return (minS);
```

Figure 1: Local Optimization

Local Optimization This technique has a number of variations; it is also referred to as *local search* [PS82]. A move is *accepted* if the adjacent state being moved to is of lower cost than the current state. If this is done repeatedly, the optimizer attains a local minimum (which is not necessarily a global minimum); see Figure 1. The sequence of moves to a local minimum from the start state is termed a *run.*

Two questions need to be answered before the algorithm in Figure 1 can be considered completely specified. (We defered the discussion of stopping criteria to Section 4.2.)

- How is the *start* state obtained?

The *start* state is the state in which we begin a run of local optimization. For the first run, the start state is the *initial* state. After a local minimum is reached, two variations on how to generate a new start state have been proposed [NSS86]. In *iterative improvement* (**II**), the start state is a state generated at random using, say, the same state generator as in **QS**. In *sequence heuristic* (**SH**), the start state is obtained by making a number of moves from the local minimum, except that this time each move is accepted irrespective of whether it increases or decreases the cost.

- How is a local minimum detected?

A state usually has a large number of neighbours. It would be impractical to exhaustively enumerate all the neighbours to verify that it is a local minimum. Instead, an

```
/* Get an initial solution */
S = initialize();
/* Set the initial temperature */
T = initialTemp();

repeat {
   repeat {
      /* Randomly selected adjacent state */
      newS = move(S);
      delta = cost(newS) - cost(S);
      if (delta ≤ 0)
        S = newS;
      if (delta > 0)
        S = newS "with probability exp(-delta/T)";
   } until ("inner-loop criterion is satisfied);

   /* Reduce the temperature */
   T = reduceTemp(T);
} until ("system has frozen");

return (minS);
```

Figure 2: Simulated Annealing

approximation based on random sampling is used. We generate and test a large number of adjacent states. If any one is of lower cost, we move to that state and start all over again. If no tested neighbour is of lower cost, the current state is taken to be a local minimum. The number of states tested before a local minimum is declared is a parameter of both **II** and **SH**, and is called the *sequence length* (denoted by *seqLength*).

Simulated Annealing (SA) This technique can also be viewed as a variation on local optimization, but it differs from **II** and **SH** in significant ways. The simulated annealing algorithm was originally derived by analogy to the process of annealing of crystals from liquid solution. Hence, the terminology of physical processes is commonly used e.g., *temperature, freezing condition*, though these physical analogies can be dispensed with (and, indeed, often are) once the algorithm and its parameters have been described. The general algorithm is shown in Figure 2.

As in other local optimization techniques, moves which decrease the cost are always accepted. However, in **SA**, moves which increase the cost can also be accepted at any time. Such moves are accepted with a probability which depends on the increase in cost entailed by making the move (*delta* in the above algorithm), and a parameter called the *temperature* (T). The exponential form of the probability function is derived from the mathematical model of the annealing process. Looking at the exponential probability function in Figure 2, it is clear that the higher the temperature, the more likely that an upward move will be accepted. Hence, the higher the temperature, the higher the fraction of moves which are accepted. Also, the probability increases with decreasing *delta*. This means that upward moves to a state of much higher cost are less likely to be accepted.

As before, a number of details need to be filled in the algorithm in Figure 2. A number of variations are possible and we will describe two important variations which we have used in our experiments. The first variation (described in [JAMS87]) is denoted by **SAJ**, and the second variation (described in [HRS86]) is denoted by **SAH**. Some of the details of these variations are omitted for brevity; the reader is referred to the cited references for a complete description.

• How is the initial temperature (denoted by T_0) determined?
In **SAJ**, T_0 is taken to be that temperature at which a significant fraction (*initProb*) of all attempted moves is accepted. A typical value for *initProb* is 0.4. To obtain T_0, one starts with the temperature being set to the cost of the initial state, and keeps doubling the temperature until a value is obtained at which *initProb* fraction of moves are accepted. In **SAH**, T_0 is taken to be $K * \sigma$, where σ is the standard deviation of the cost distribution (estimated by some initial sampling). A reasonably high value for K is 20.

• How does reduceTemp work?
In **SAJ**, the current temperature T is multiplied by a fixed reduction factor called *tempFactor*. A typical value for *tempFactor* is 0.95. In **SAH**, the reduction factor is given by the value of the function $\max(0.5, \exp(-\lambda T/\sigma))$, where a typical value for λ is 0.7. The lower bound of 0.5 is used to prevent precipitous annealing at high temperatures.

• What is the inner-loop criterion?
The inner-loop criterion determines the number of times that the inner loop is executed. This number is called the *chain length*. In **SAJ**, the chain length is given by *sizeFactor* * N, where *sizeFactor* is a parameter. Here the chain length is fixed throughout the annealing. In **SAH**, the chain length is dynamically adjusted. The idea is to allow "... the establishment of a steady-state probability distribution of the accessible states" [HRS86]. We omit the details of how this adjustment is done.

• What is the freezing condition?
The freezing condition determines when the annealing process stops. The intuition is that the temperature is small enough that the probability of accepting upward moves is negligible, and very few downward moves are discovered. In **SAJ**, the freezing condition consists of two tests. The first test is whether there has been any improvement over the best solution value during the last 5 temperatures. If this test fails, then a check is made to see whether the percentage of accepted moves at the current temperature exceeds a parameter denoted by *minPercent*. If neither test is satisfied, the system is said to have "frozen".

In **SAH**, the difference between the maximum and minimum costs among the accepted states at the current tem-

perature is compared with the maximum change in cost in any accepted move during the current temperature. If they are the same, the system is declared frozen since "... apparently all the states accessed are of comparable costs, and there is no need to use simulated annealing [any further]" [HRS86].

4 Application to Query Optimization

We now discuss how we adapted the general algorithms described in Section 3 to the optimization of large join queries. As explained in Section 2, the algorithms need to determine a good outer linear join processing tree (OLJT). This is equivalent to determining an optimal (or, a good) ordering i.e., permutation, of the relations. The search is restricted to the space of valid OLJTs (or, equivalently, valid permutations). In this section, the permutation notation will be used to represent the corresponding join tree. A state, then, is a valid permutation. The join processing cost model is used to estimate the cost of a state.

4.1 Move Set

We now discuss the different kinds of moves. Let S be the current state.

S = (... i ... j ... k ...)

The two kinds of moves that we use are *Swap* and *3Cycle*.

- *Swap*
  Select two distinct relations, say, i and j, at random. Check if interchanging i and j results in a valid permutation. If so, the move consists of swapping i and j to get the new state newS.

  newS = (... j ... i ... k ...)

- *3Cycle*
  Select three distinct relations, say, i, j and k, at random. The move consists of cycling i, j and k: i is moved to the position occupied by j, j is moved to the position occupied by k, k is moved to the position occupied by i. We first check if the resulting permutation is valid, and if that is the case, we get the new state newS.

  newS = (... k ... i ... j ...)

Our move set is then (*Swap*, *3Cycle*, α), $\alpha \in [0, 1]$, where α is the frequency with which *Swap* is selected (clearly, $1 - \alpha$ is the frequency of *3Cycle*). The parameter α needs to be adjusted for each technique.

In most work on combinatorial optimization problems, a move is a small perturbation in the state to get another state. This is satisfied by our choice of moves. It is also desirable that the cost of the new state be incrementally computable from the cost of the current state in order that testing the move is not expensive. In the moves that we use, we usually do not have to traverse the entire join tree to compute the new cost. These two kinds of moves have been used in tackling other combinatorial optimization problems, and more complex moves can be regarded as composed of sequences of these moves.

4.2 Stopping Criteria

An important question in the design of the algorithms for query optimization is that of stopping criteria. One simple criterion is obtained by specifying a maximum time limit, and terminating the optimization process if it exceeds the time allowed. This criterion is useful independent of the other stopping criteria being used. Hence, in addition to the other criteria, we use the criterion of a maximum time limit in all our experiments. Indeed, the maximum time allowed is a parameter, just like other parameters of an algorithm.

One could stop earlier if one reaches a global minimum. The problem is that of identifying a global minimum. One must remember that the size of the problems is such that exhaustive search to obtain the global minimum for the benchmark queries is impractical. The best we can do is to estimate a *lower bound* on the cost of a global minimum, and use this in the stopping criterion. One natural way to use the lower bound is to stop when the optimizer obtains a solution whose cost is sufficiently close to the lower bound. In our experiments, we define "sufficiently close" to be within twice the estimated lower bound.

To use this stopping criterion, we need to estimate a lower bound. In combinatorial optimization problems like the Traveling Salesman Problem (TSP), the theory has been sufficiently developed to permit very good estimates of the lower bound. This is not the case in query optimization. We currently use comparatively crude estimates of the lower bound. Essentially we sum the costs of all the processing which is independent of the size of the intermediate results e.g., the costs of scanning the relations or the costs of creating the hash tables.

Since we do not know how good the lower bound is, we cannot always assess, in an absolute sense, the quality of the solution produced by an algorithm. Hence, one cannot make very definitive statements about an algorithm in isolation. However, comparisons between algorithms are not affected. Also, the use of a bad estimate of the lower bound in the stopping criterion only means that the algorithm may run longer than needed; on the average, the quality of the solution produced is not affected.

4.3 Is the Algorithm Good?

It remains to decide on a measure of the goodness of an algorithm. One often needs such a single measure e.g., it is needed for the factorial experiments discussed in Section 5. From a practical point of view, we regard an algorithm as being good if it *performs well on the average, and very rarely performs poorly.* We wish to translate this somewhat vague specification into a quantitative measure. In order to compare the performance of the algorithms on different queries, we need to scale the cost of the solutions obtained. The scaling is performed by dividing by the cost of the

best solution obtained. Thus, the solution quality is a dimensionless number greater than or equal to 1.

It would be easy enough to simply compare the means of the scaled solution costs. However, this would not necessarily reflect the criterion stated in the above paragraph as the following (hypothetical) example shows.

Example: Two algorithms A_1 and A_2 are being compared. The same queries were optimized using both A_1 and A_2, and the following results were obtained (we show the scaled solution costs along with the percentage of the queries having these costs).

A_1: 60% - 1; 30% - 2; 10% - 10
A_2: 40% - 1; 40% - 2; 16% - 5; 4% - 10

If one computes the simple means

$\text{mean}(A_1) = 2.20$; $\text{mean}(A_2) = 2.40$

then it would appear that A_1 was better than A_2. However, according to the criterion we have specified, we would actually prefer A_2.

□

To resolve this problem, we define the notion of an outlying value. An *outlying value* is a final solution cost (obtained by some algorithm) which is much higher than the best solution cost. Intuitively, an algorithm performs poorly on a particularly query if the solution cost obtained by the algorithm is an outlying value. In our experiments we define a solution cost to be an outlying value if it is at least 10 times the best solution cost. Note that there is no problem in using the best solution cost as a standard because these are comparative experiments, and the analysis is performed after all the algorithms have optimized the queries.

A better measure than the simple mean is obtained as follows. First we compute the mean *excluding* the outlying values (denote this "trimmed" mean by a). We compute the count of the outlying values (denote this by b). Our measure of goodness of the algorithm is then $a + b$. The reason we just count the outlying values is that once a solution is considered poor, we are not much interested (from a practical point of view) in how poor it is. For the same reason, we do not include the outlying values in the computation of the mean; they would skew the mean too much; the mean is not a very robust statistic. The trimmed mean a tries to capture the performance of the algorithm "on the average", and b tries to quantify how often the algorithm "performs poorly."

We have simplified things somewhat; we actually add the square root of b to a. An explanation of this *variance stabilizing transformation* (and other such transformations) can be found in [BHH78]. Hence, our measure is $a + \sqrt{b}$. Using this new measure in our example, we find

$a_1 = 1.33$; $b_1 = \sqrt{10} = 3.16$
$a_2 = 2.08$; $b_2 = \sqrt{4} = 2$
$a_1 + b_1 = 4.49$; $a_2 + b_2 = 4.08$

Now $a_1 + b_1 > a_2 + b_2$, and our measure agrees with our intuition that A_2 is better than A_1.

4.4 Query Generation

To compare the various algorithms in a comprehensive manner using the statistical methods discussed in Section 5, we need to generate a large number of queries. This is done as follows. Distributions for various parameters of the queries are specified. Values for these parameters are then obtained using random numbers distributed accordingly. Different queries are generated by specifying different initial seeds. This enables us to obtain a large number of "random" queries.

N, the number of joins, is allowed to take values 10 through 100 (the number of joining relations is $N + 1$). The join graph is generated as follows. In the initial permutation (1 2 3 ... N $N + 1$), a connected join graph is obtained by using N joins to make the permutation a valid permutation. Observing this constraint, the relation to join with a given relation is chosen at random. Then, upto N additional joins are assigned (again at random). The joining attributes are selected at random.

The *relation cardinality* is the number of tuples in a relation. Each relation can have selection predicates which restrict the tuples of the relation which participate in joins with other relations. The number of *distinct values* in a join column is an important factor in determining the size of intermediate results. Finally, indexes can be used as efficient access paths.

The features which characterize individual relations are distributed as follows:

- Relation Cardinalities
  [10, 100) - 20%; [100, 1000) - 64%;
  [1000, 10000) - 16%
- Selections
  The number of selection predicates per relation ranges from 0 to 2. The selectivities of the selection predicates were chosen randomly from the following list:
  0.001, 0.01, 0.1, 0.2, 0.34, 0.34, 0.34,
  0.34, 0.34, 0.5, 0.5, 0.5, 0.67, 0.8, 1.0
- Distinct Values in join columns (as a fraction of the relation cardinality)
  (0, 0.2] - 75%; (0.2, 1) - 5%; 1.0 - 20%
- Indexes
  About 25% of the columns involved in join predicates were selected to have indexes (the actual columns were chosen at random)

We model a distribution of the relation cardinalities where we have a small but not insignificant number of both small and large relations, and a majority of the relations are of medium size. The available selectivities were chosen so that the selectivities of 1/3 and 1/2 (estimates used by many query optimizers like System/R) were the most common, and there are a few small and large selectivities. In deciding on the distribution of distinct values, we took into account that columns with unique values (e.g., key columns) are often present. The remaining columns are assumed to have a much smaller number of distinct values. This works out to about 10% of the relation cardinality on

the average. Again, this is an estimate often used. We have not seen any studies on the average number of indexes and chose the figure above as a reasonable one.

5 Experimental Comparison of Algorithms

The different algorithms were coded in C, and the experiments were run on very lightly loaded HP 9000/350 workstations. The workstation is based on a 25 MHz 68020 processor, and is approximately a 4 MIPS machine. Note that the optimizer programs are completely CPU bound; memory requirements are negligible compared to the main memory of the workstations.

For the experiments on tuning the parameters of the algorithms, we used 50 different queries for each of $N = 10, 20, 30, 40, 50$, giving a total of 250 different queries. Each algorithm was run twice on each query (using different initial seeds), thus giving two *replicates* per query which were then averaged. For the experiments on comparing the algorithms, we used 50 different queries for each of $N = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100$, giving a total of 500 different queries. We obtained four replicates for each algorithm per query. The entire set of experimental runs took about six weeks with the programs running most of the time.

5.1 Tuning of Parameters

Before comparing the different algorithms, we need to ensure that the parameters of the algorithms are set to "good" values. An example of a parameter (which happens to be common to all the algorithms except **QS**) is α, the frequency with which *Swap* is selected. Parameters particular to the algorithms are:

II, SH : *seqLength*
SAJ : *tempFactor, initProb, sizeFactor, minPercent*
SAH : *K, size, minAccept, M* (see [HRS86])

To obtain reliable values for these parameters, we use the methodology of factorial experiments [Dav78], [BHH78]. To be more precise, we use 2^n factorial and fractional factorial experiments, where n is the number of parameters. Usually, each parameter (or *factor*) can take on a number of values, and it is very expensive or time consuming to try all possible combinations of all possible values of the factors. The idea in 2^n factorial experiments is to consider, in one experiment, two levels (called *low* and *high*) of each factor, and try out all combinations of these levels.

Statistical theory enables us to estimate the main effects of the parameters and their interactions. We then vary the values of the parameters in the direction of increasing benefit. If the parameter has no significant effect, we reduce the range i.e., we decrease the separation between the low and the high levels. In fractional factorial experiments, we do not try all combinations of the levels, but only a *fraction* of these combinations in such a way that we can still estimate the effects of interest to us. Details of this methodology and its advantages over the more common "one parameter at a time" method are explained in the references cited earlier. An example illustrating the use of factorial experiments is given in [Swa87b].

5.1.1 Time is a Factor

In addition to the parameters mentioned above, we also included *time* as a factor. By time we mean the maximum time allowed for one optimization run; this is used in the stopping criterion discussed in Section 4. This enables us to determine if a particular algorithm shows no significant change in the quality of the solutions produced beyond a certain time limit (we say that the algorithm "saturates" at this time limit). The performance of the algorithm at the saturation time limit approximates its performance if it were potentially given unlimited time.

Introducing time as a factor allows us to discover and study interactions between time and other factors (if such interactions exist). The time limit for a query is proportional to N^2, with the constant factor changing for different time limits. When we mention time limits we will specify it for a single value of N; the corresponding time limits for other values of N can be easily deduced. For example, if the time limit at $N = 50$ is 7.5 minutes, the corresponding time limit at $N = 100$ is $(\frac{100}{50})^2 * 7.5 = 30$ minutes.

5.1.2 Results

The results of tuning the parameters using factorial experiments are given below (all saturation times are in minutes and are given for $N = 50$).

- **II**: *seqLength* $= N$; *time* $= 7.5$
- **SH**: *seqLength* $= N$; *time* $= 7.5$
- **SAJ**: *sizeFactor* $= 1$; *tempFactor* $= 0.975$; *minPercent* $= 2$; *initProb* $= 0.4$; *time* $= 7.5$
- **SAH**: $K = 20$; *size* $= 2N$; *minAccept* $= 2N$; $M = N^2/2$; *time* $= 7.5$
- **PW**: *time* $= 5$
- **QS**: *time* $= 7.5$

The parameter α takes the value of 0.5 for all the algorithms. Actually, no particular value of α proved better than any other value. We arrived at the value of 0.5 by our method of decreasing the range whenever a parameter had no significant effect.

5.2 Comparison of Algorithms

The maximum saturation time is 7.5 minutes at $N = 50$. This time is 30 minutes at $N = 100$. We could compare the algorithms only at this saturation time. However, it is possible that the ordering among the algorithms may change with different time limits. Hence we decided to

compare the algorithms for a range of time limits upto the time limit of 30 minutes. The time limits we experimented with are 1 minute, 2 minutes, 10 minutes, and 30 minutes (all these times are specified at $N = 100$).

To compare the algorithms we used the techniques of Analysis of Variance (ANOVA) [DW83]. The idea is to run the algorithms on a large number of different queries. Now, we wish to see if there is any significant difference among these algorithms. ANOVA checks this by comparing the spread *among* the algorithms with the spread *within* an algorithm. If these are comparable, then no significant difference can be detected. If not, the algorithms differ in their performance.

The spreads are compared using the following standard F–test on the ratio of two mean sum of squares (MSS). Let MSS_a measure the spread *among* the algorithms, and MSS_b measure the spread *within* the queries optimized by an algorithm. Then, the probability of the ratio MSS_a/MSS_b (this will be denoted by F_{ab} below) attaining or exceeding this value according to the F distribution is obtained (this probability is expressed as a percentage below). The lower the probability the more significant is the difference among the algorithms. As in the factorial experiments, percentages higher than 10% indicate *statistically* insignificant differences.

If the test indicates that there exists a significant difference among the algorithms, we would like to group the algorithms according to how well they perform i.e., algorithms which do not differ among themselves would be in the same group. The way we do this is to examine the mean performance of the algorithms to see if we can identify any groups. Note that we are using means of the scaled solution values i.e., the solution values are divided by the best solution values. Then we verify that our grouping is correct by checking that

- the algorithms in a group do not differ among themselves
- when an algorithm from one group is included in another group, there is a significant difference among algorithms in the new group

We use the same F–test for identifying significant differences (only now we apply the test to subsets of the set of all algorithms). We can then order the groups (and the algorithms within a group) according to their means. An example illustrating this methodology is given in [Swa87b]. Another good example is to be found in [NMF87].

5.2.1 Ordering Among Algorithms

In Table 1, we present the results of our comparative experiments. At each time, we identify the algorithm groups by enclosing them in brackets. The algorithms which perform better come earlier in the sequence. We also give the means of the scaled solution costs; the cost have been scaled by dividing by the best solution costs obtained at 30 minutes. Care should be taken in interpreting these means as was noted in Section 4.3.

Time	Groups and Means
1 minute	[II] [SAH, SH, QS, SAJ] [PW] [1.62] [2.74, 2.92, 2.95, 3.09] [3.71]
2 minutes	[II] [SAH, QS, SH, SAJ] [PW] [1.62] [2.69, 2.74, 2.83, 2.88] [3.38]
10 minutes	[II] [SAJ, QS] [SAH, SH, PW] [1.34] [2.27, 2.35] [2.47, 2.56, 2.69]
30 minutes	[II] [SAJ] [QS] [PW, SH, SAH] [1.17] [1.82] [2.15] [2.34, 2.39, 2.45]

Table 1: Comparison of Algorithms

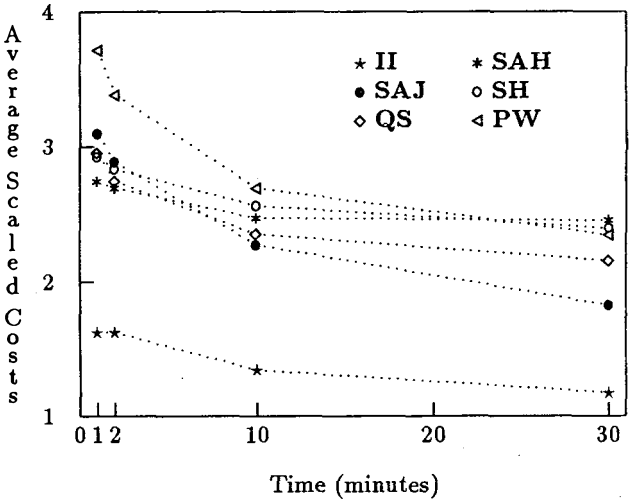

Figure 3: Sensitivity to Time

Clearly, *iterative improvement* (II) is superior to all the other algorithms over all time limits. Using more complex algorithms like simulated annealing does not result in better performance. In fact, the performance of II at 1 minute (mean = 1.62) is better than the performance of SAJ (mean = 1.82) at 30 minutes. Note that SAJ is the second best algorithm at 30 minutes. The *sequence heuristic* (SH) and *perturbation walk* (PW) are clearly inferior at all the time limits.

One possible explanation of these results is that the solution space has a large number of local minima, with a small but significant fraction of them being deep local minima. II can traverse large regions of the search space and thus stands a good chance of finding one of the deep minima. QS too traverses large portions of the search space, but II does better because it always ends up in a local minimum. PW and SH do badly because they do not travel far from the starting state. The simulated annealing algorithms do not travel as much as II because they never make total random moves once the initial state is chosen. However, they travel more than PW and SH as they can accept cost increasing moves. We are working towards characterizing the search space better. We hope

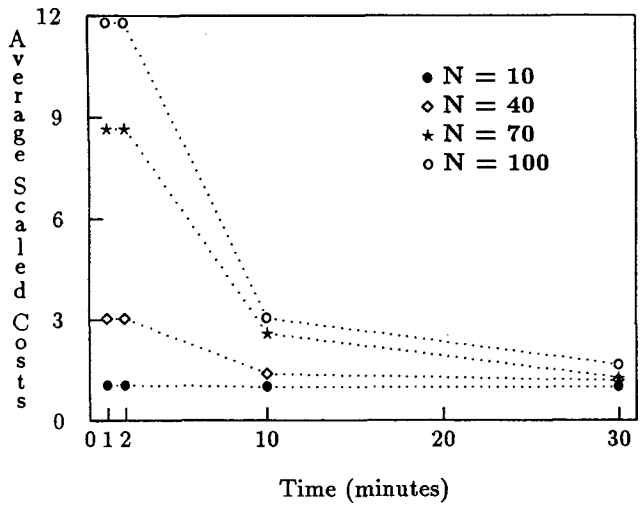

Figure 4: Performance of **II** with Time

then to come up with a better explanation of these results.

However, as the time given increases, simulated annealing can improve its performance, since it can travel further. Thus, the simulated annealing algorithm described in [JAMS87] (**SAJ**) moves from being the second worst algorithm at 1 minute to being the second best algorithm at 10 and 30 minutes. It is not clear why the simulated annealing algorithm described in [HRS86] (**SAH**) slips in the ranking.

We also found from the data that at 30 minutes less than 10% of the best solutions were outlying values i.e., greater than ten times the lower bound. Also, the means of the best solutions which were not outlying values were close to twice the lower bounds. Note that we halt the algorithms once they achieve a cost which is less than twice the lower bound. All this means that at the saturation time algorithms like **II** perform reasonably well in solving the large join query optimization problem (LJQOP).

5.2.2 Effect of Time

In Table 1, we find that as the time limit increases, the algorithms other than iterative improvement split into smaller groups. This means that the performances of some algorithms are more sensitive to time than others. To illustrate this, we use the data in Table 1 to draw Figure 3. We can see in the figure how the groups split up as some algorithms improve more rapidly with time than others. All the algorithms improve significantly going from 1 minute to 10 minutes; except for **SAJ**, the improvement in going from 10 minutes to 30 minutes is not so marked.

We investigate **II** further since it proves to be clearly superior. In Figure 4, we show how the average scaled costs obtained by iterative improvement (**II**) changes with time for $N = 10, 40, 70, 100$. We graphs the costs at 1 minute, 2 minutes, 10 minutes, and 30 minutes. In all cases, the costs have been scaled using the best solutions obtained at the time limit of 30 minutes. We now see that the large improvement in performance when time is increased from 1 minute to 10 minutes holds mainly for larger N; for $N = 10$, there is very little improvement. We observe that for all N there is not much improvement when time is increased from 10 minutes to the saturation time. This is an interesting fact which may be used when optimizing queries which are expected to be executed only a few times. In that case, one may obtain sufficiently good solutions at times much less than the saturation time.

6 Discussion

The large join query optimization problem (LJQOP) is a hard combinatorial optimization problem. Such problems have been tackled using general techniques such as those based on local search. Another approach is to use heuristics, such as "divide and conquer" which can drastically reduce the search space. In this paper we have investigated the first approach to LJQOP. We have described the algorithms of perturbation walk, quasi–random sampling, iterative improvement, sequence heuristic, and simulated annealing. We showed how these techniques can be adapted to LJQOP.

We described how these algorithms can be tuned using factorial experiments. By using time as a factor, we were able to reliably determine the saturation time, that is the limit beyond which increasing time shows no significant improvement in the performance of the algorithm. Also, the fact that the parameter (α) which gives the fraction of *Swap* moves to *3Cycle* moves has no effect leads us to hypothesize that our choice of simple moves is adequate.

We then described how we used analysis of variance (ANOVA) to compare the different algorithms. Iterative improvement is superior to all the other algorithms at all the time limits. Also, it seems that simulated annealing *by itself* is not an useful technique for LJQOP. We suggested that one possible explanation of our results of the comparison of the algorithms is that the search space has a large number of local minima, with a small but significant fraction of them being deep local minima, and **II** does well because it can traverse large regions of the search space.

We also showed that at the saturation time over 90% of the best solutions were quite close to our target of twice the lower bound. The good quality of the solutions obtained indicates that we can do reasonably well in solving LJQOP using just these general algorithms. We also showed that the performances of all algorithms except **SAJ** improve rapidly up to a certain time limit beyond which much smaller improvements are obtained. This time limit is less than the saturation time. This can be used to help decide how much time to spend on optimizing a query. If a query will be used only a few times, one can obtain sufficiently good solutions using a time limit smaller than the saturation time.

Our work can be extended in various ways. In future, we intend to include join methods other than the hash join method. We also plan to incorporate other cost mod-

els e.g., a cost model for disk–based query processing. In addition, we will look at large join queries generated by strategies which try to introduce "clusters"; these may model a number of useful kinds of queries.

We intend to investigate the other important approach to combinatorial optimization problems viz., use of heuristics. It would be interesting to compare these heuristics to the algorithms we have investigated. Finally, a combination of general algorithms and heuristics may prove to be useful; this also needs to be investigated.

Acknowledgements

The first author acknowledges Prof. Gio Wiederhold, Peter Lyngbaek and Marie-Anne Neimat for their encouragement and support, and for helping by way of discussions and comments on earlier drafts. He also thanks Tim Read for advice regarding statistical theory and practice. Arun Swami is supported by Hewlett–Packard Laboratories under the contract titled "Research in Relational Database Management Systems" and, earlier, was supported by DARPA contract N00039-84-C-0211 for Knowledge Based Management Systems. Anoop Gupta is supported by a faculty award from Digital Equipment Corporation.

References

[BHH78] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters.* John Wiley and Sons, 1978.

[Dav78] O. L. Davies, editor. *The Design and Analysis of Industrial Experiments.* Longman Group Limited, 2nd edition, 1978.

[DKO*84] D. J. Dewitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[DW83] S. Dowdy and S. Wearden. *Statistics for Research.* John Wiley and Sons, 1983.

[FBC*87] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An Object–Oriented DBMS. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[HRS86] M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An Efficient General Cooling Schedule for Simulated Annealing. In *Proceedings of the 1986 IC CAD Conference*, pages 381–384, 1986.

[IK84] T. Ibaraki and T. Kameda. Optimal Nesting for Computing N-relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, October 1984.

[IW87] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9–22, 1987.

[JAMS87] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation (Part I). June 1987. Draft.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the Twelth International Conference on Very Large Data Bases*, pages 128–137, Kyoto, Japan, 1986.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[NMF87] R. E. Nance, R. L. Moose, and R. V. Foutz. A Statistical Technique for Comparing Heuristics: An Example from Capacity Assignment Strategies in Computer Network Design. *Communications of the ACM*, 30(5):430–442, May 1987.

[NSS86] S. Nahar, S. Sahni, and E. Shragowitz. Simulated Annealing and Combinatorial Optimization. In *Proceedings of the 23rd Design Automation Conference*, pages 293–299, 1986.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[SAC*79] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979.

[Swa87a] A. Swami. A Cost Model For Memory Resident Databases. April 1987. Computer Science Department, Stanford University.

[Swa87b] A. Swami. *Optimization of Large Join Queries.* Technical Report, Software Technology Laboratory, Hewlett–Packard Laboratories, November 1987. Report STL-87-15.