Genetic Algorithm for the Multiple-Query Optimization Problem

Murat Ali Bayir, Ismail H. Toroslu, and Ahmet Cosar

***Abstract*—Producing answers to a set of queries with common tasks efficiently is known as the multiple-query optimization (MQO) problem. Each query can have several alternative evaluation plans, each with a different set of tasks. Therefore, the goal of MQO is to choose the right set of plans for queries which minimizes the total execution time by performing common tasks only once. Since MQO is an NP-hard problem, several, mostly heuristics based, solutions have been proposed for solving it. To the best of our knowledge, this correspondence is the first attempt to solve MQO using an evolutionary technique, genetic algorithms.**

***Index Terms*—Database query processing, genetic algorithms (GA), heuristics techniques, multiple-query optimization (MQO).**

I. Introduction

Multiple-query optimization (MQO) is a well-known database research problem, and it has been studied by the database community since the 1980s. The goal of MQO is to reduce the execution cost of a set of queries by performing their common tasks only once. A query may have more than one solution plan. Traditional query optimization considers a single query at a time and tries to generate the most efficient solution by examining and finding cheaper alternative plans. To solve several queries simultaneously, and to identify their common tasks, the entire set of alternative plans of these queries must be processed together, because plan expensive tasks might result in more sharing with other queries and yield a better solution to the MQO problem.

Query optimizers for relational databases use relational algebra (RA) [1] for internal query representation. Most of the research on databases has focused on select-project-join queries, and the experiments in this correspondence have also been limited to those operations. It is assumed that the set of alternative plans will be generated by a multiple-query optimizer, as described in [2], or by postprocessing of single-query optimizer-generated plans to obtain *more sharable* alternative plans, as described in [3] and [4]. Once the common tasks and their execution time estimates are provided, our optimization algorithm is applicable for the complete set of RA operations and vendor-specific extensions.

Single query optimization aims to minimize the time required to calculate the output for a given query. This problem has been addressed in great detail by database researchers since the mid-1970s [5]. Recent developments on this problem can also be found in [6]. Because of the large number of alternative join methods and join orders, state-of-the-art optimizers search only a subset of the possible query plans, to be specific, left-deep (nonbushy) join trees, and generate a single execution plan in the form of a tree with RA operations as its internal nodes and base relations as leaf nodes.

Even though the above approach is acceptable for single queries, we are faced with the problem that in MQO a suboptimal plan may be preferred for a query if it results in more common tasks with the other queries in the group and thus yields a lower cost global plan for the query group. We are assuming that common tasks have, already, been determined by the first phase of MQO, alternative plan generation.

Manuscript received January 26, 2005; revised May 27, 2005. This paper was recommended by Associate Editor J. Lazansky.

The authors are with the Computer Engineering Department, Middle East Technical University, 06531 Ankara, Turkey (e-mail: ali.bayir@ceng.metu.edu.tr; toroslu@ceng.metu.edu.tr; cosar@ceng.metu.edu.tr).

Digital Object Identifier 10.1109/TSMCC.2006.876060

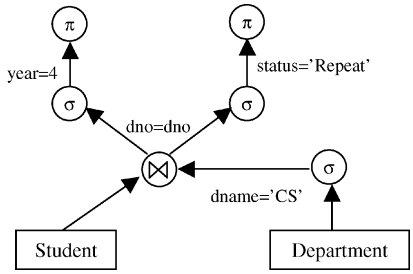

Fig. 1. Multiplan 1 for evaluating both Q_1 and Q_2.

One of the first formulations of MQO can be found in [7]. Under this formulation, the MQO problem can be divided into two phases. The first phase of the MQO performs the identification of common tasks among a set of queries and prepares a small set of alternative plans for each query. This set of alternative plans will be used in the second phase of MQO for the selection of exactly one plan for each query. The second phase generates a global execution plan that will produce the answers for all the queries when it is executed.

The first phase of MQO has been addressed in a recent paper [8], and it has been shown that for small problem sizes (up to ten queries), near-optimal global plans can be generated very efficiently. Also, it is possible to generate a small set of alternative plans for a query, given a set of queries (with which we are trying to maximize shared tasks), while preserving the quality of plans and to obtain multiplans always within 20% (on an average 12%) of the optimal global plan. Another interesting observation is that, as the number of queries in a MQO problem instance increases, the multiplans generated still remain close to optimal.

Usually, the second phase of MQO is the more time-consuming phase of the problem. In the second phase, an efficient global execution plan for the set of queries, whose plans and common tasks between these plans have already been determined in the first phase, is generated. In [7], the second phase of MQO is shown to be NP-complete. By using the formulation of [7], the second phase of MQO can be studied independently from the first one. Some research works dedicated to efficiently solving the second phase are [9]–[11]. All of these use heuristic techniques, such as A^*.

To illustrate the task sharing on how MQO exploits common tasks and generates a global plan, let us assume that we want to execute the following queries together.

Q1: Find the names of senior students in the computer science department.

Q2: Find the various id of "repeat" students in the computer science department.

Only the select operation *dname = "CS"* on the *Department* relation is common to both the queries. It is possible to share the join operation between *Department* and *Student* relations, as well, but, for this, we must delay the selection operations "*status = repeat*" and "*year* $= 4$."

It is possible to generate two alternative plans for the execution of these queries. Figs. 1 and 2 describe these two plans for the queries above. The important point to recognize in this example is the delaying of the selection operations, which is just the opposite of the conventional policy of performing selections as early as possible in single-query optimization. Although delaying selections increases the cost of evaluating each query individually, because of the one-time execution of a more expensive common task instead of two or more cheaper tasks (with a longer total execution time) this becomes a good technique to obtain alternative plans for individual queries, and we can use these alternative plans as input for the MQO phase.

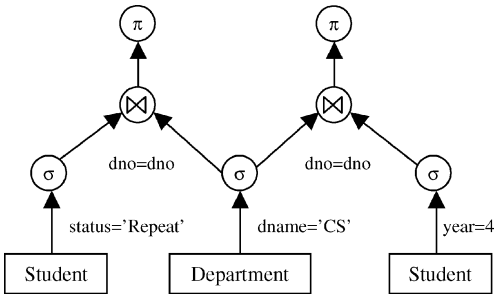

Fig. 2. Multiplan 2 for evaluating both Q_1 and Q_2.

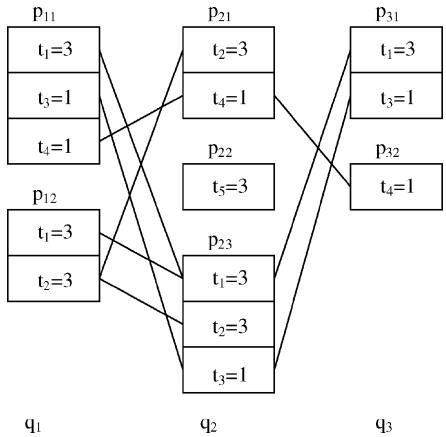

Fig. 3. MQO problem with three queries, seven plans, and five tasks.

Although earlier works have aimed at finding optimal solutions to the MQO problem, more recently, research on MQO shifted to efficiently generating alternative plans that maximize shared operations [10], [12]. The work of Roy *et al.* [10] describes a greedy heuristic for generating promising alternative plans. This solution can be described as a real-time solution to the MQO problem since it uses already generated results during the query execution to generate alternative plans by selecting plans such that the amount of shared tasks in those plans will be as high as possible. Dalvi *et al.* [12] extend the work of Roy *et al.* [10] further and use the idea of pipelining intermediate shared results instead of materializing them to a hard disk. These recent works' MQO formulation is slightly different than the one in [7]. As in [7], we also assume that at the beginning all promising alternative plans have been generated and shared tasks are identified. That is, the first phase has been completed, and our focus will be on developing an efficient solution to the second phase of the MQO problem. On the other hand, unlike [7], [9], and [11], all of which finds the optimum solution to the second phase of the MQO problem, we aim to use metaheuristic techniques to find "good" solutions for larger problem instances in a more efficient way. State-of-the-art applications of the MQO problem to the solution of real database problems require solution of much larger problem instances with dozens, possibly hundreds, of queries, which are far beyond the reach of currently available heuristic search algorithms.

As one of these applications areas, there has been a lot of recent interest in using multiquery optimization techniques in the context of materialized views and data warehouses. In [13], multiquery optimization is used to identify and exploit common subexpressions for materialized view maintenance. Another possibility is to materialize views such that it would improve query response times by considering frequently used common subexpressions as candidates for view materialization; a very detailed survey of previous work on answering queries using views is given in [14]. An extension of multiquery optimization to queries with aggregate operators has also been proposed in [15] and shows the great potential MQO has for expensive queries. The maintenance and verification tasks on relational databases can be improved by employing multiquery optimization techniques, as discussed in [16], as well.

In this correspondence, a popular metaheuristics technique—a genetic algorithm (GA)—is used to solve the second phase of the MQO problem. As in several NP-complete optimization problems, the evolutionary approach of the GA is also expected to produce a feasible solution for MQO in a computationally acceptable time.

This correspondence is organized as follows. Section II gives the formulation of the MQO problem used in this correspondence. Section III describes the A* formulation used in [7] and [9]. Section IV introduces the GA metaheuristic technique and presents GA formulations for the MQO problem. Section V presents and analyzes the experimental performance results and their comparisons. Finally, a conclusion is given in Section VI.

II. MQO Formulation

In an MQO problem, a set of (N) queries $\{q_1, q_2, \ldots, q_N\}$ are optimized together. Each query q_i has a number of (M_i) possible solution plans, and each plan of a query contains a set of tasks (RA operations), which when executed in a certain order will produce the answer for the query. Each task also has an associated cost, and for convenience we will represent the cost value by a positive integer number. Alternative plans of a query, and other queries in the query set, may contain the same task. Therefore, in solving the MQO problem, the aim is to determine a set of tasks, with minimal total cost, that contains all the tasks of at least one plan of each query (or alternatively determine a set of plans, containing one plan for each query, with the minimum total cost, in which shared tasks' costs are considered only once during the computation).

We will use the example in Fig. 3 to illustrate the MQO problem and the solution techniques used in this correspondence. In the figure, straight lines are used to identify common tasks in the plans of different queries. The parameters of this example are as follows.

- There are three queries $(N = 3)$ represented as q_1, q_2, and q_3.
- Altogether, there are five different tasks that can be used to answer all three queries. Tasks are named as t_i. The costs of the tasks are represented as c_i and they are as follows. Task t_1 with cost $c_1 = 3$, task t_2 with cost $c_2 = 3$, task t_3 with cost $c_3 = 1$, task t_4 with cost $c_4 = 1$, and finally, task t_5 with cost $c_5 = 3$.
- There are seven different plans: two for answering the first query, three for answering the second query, and two for answering the third query. Plans are named as p_{ij} representing the jth plan of query q_i. The plans of the first query are $p_{11} = \{t_1, t_3, t_4\}, p_{12} = \{t_1, t_2\}$. The plans of the second query are $p_{21} = \{t_2, t_4\}, p_{22} = \{t_5\}, p_{23} = \{t_1, t_2, t_3\}$. Finally, the plans of the third query are $p_{31} = \{t_1, t_3\}, p_{32} = \{t_4\}$.

III. A* Algorithm for MQO

Among previous heuristic solutions, [9] and [11] are known as the best MQO algorithms in the literature. Both of them are similar A* heuristic algorithms. As in a typical A* heuristic technique, the search starts with an initial state, which has all *null* values denoting that no plan has yet been selected for any queries. Then, in the intermediate states, plans are assigned to the queries, such that at each level of the search tree, one additional query gets assigned a plan. We represent the state after selecting plans for queries q_1 through q_k as $S_k = \langle p_{1s1}, \ldots, p_{ksk}, \text{null}, \ldots, \text{null}\rangle$. When all queries are assigned with a plan, it is called a solution state.

State	Estimated Cost	Action
<-,-,->	3.83	expanded
<p_{11},-,->	8	in expansion list
<p_{12},-,->	7	expanded
<p_{12},p_{21},->	7	expanded
<p_{12},p_{22},->	10	pruned
<p_{12},p_{23},->	7	expanded
<p_{12},p_{21},p_{31}>	8	not-solution
<p_{12},p_{21},p_{32}>	7	solution
<p_{12},p_{23},p_{31}>	7	solution
<p_{12},p_{23},p_{32}>	8	not-solution

Fig. 4. Execution of the A* heuristic algorithm.

We define the set of tasks in the selected plans for queries q_1 through q_k as follows:

$$t_{\text{sel}} = \bigcup_{(1\le i\le k)\in \text{and } (1\le j\le P_i)} p_{ij}$$

Let m_i be the number of queries among the remaining set of queries without an assigned plan, with an alternative plan containing the task t_i. Then, the estimated cost of task t_i is defined as

$$\text{est\_cost}(t_i) = \frac{\text{real\_cost}(t_i)}{m_i}, \qquad \text{if } t_i \notin t_{\text{sel}}, \text{ and zero otherwise.}$$

Then, the estimated cost of plan p_{ij} is

$$\text{est\_cost}(p_{ij}) = \sum_{t_i \in p_{ij}} \text{est\_cost}(t_i).$$

Finally, let P_i represent the number of alternative plans for the query i; then, the heuristic function used in A* algorithm is as follows:

$$h_s(S_k) = \sum_{t_x \in t_{\text{sel}}} \text{real\_cost}(t_x) + \sum_{k<i\le Q} \min(\text{est\_cost}(p_{i1}), \ldots, \text{est\_cost}(p_{ip_i})).$$

Further improvements are also possible on the above heuristic function. In [7], to reduce the estimation error in the heuristic the plan cost estimation function defines and experimentally evaluates six alternative query ordering heuristics for determining the best order of alternative plan assignment for each query in the MQO query set. This order is precalculated before the MQO search begins and remains static throughout the A* search. It is reported in [7] that the heuristic, which orders queries in decreasing average query costs, gives the best performance. The same result is also verified in [11] once more. Therefore, in our experiments we use this query ordering heuristic for comparison of performance results.

Fig. 4 shows the execution of the A* heuristic for the example given in Section II. Initial estimated cost is determined as 3.83. Choosing the minimum cost plans for each query is an upper bound of the total cost for evaluating all three queries. For our example, this value is 8. Note that the upper bound might be less than the summation of the costs of minimum costly plans due to common tasks. If an expanded state that represents a (partial or complete) solution that has an estimated cost greater than 8, then that branch of the search tree is pruned. Fig. 5 also shows the corresponding search tree generated for solving the same example MQO instance.

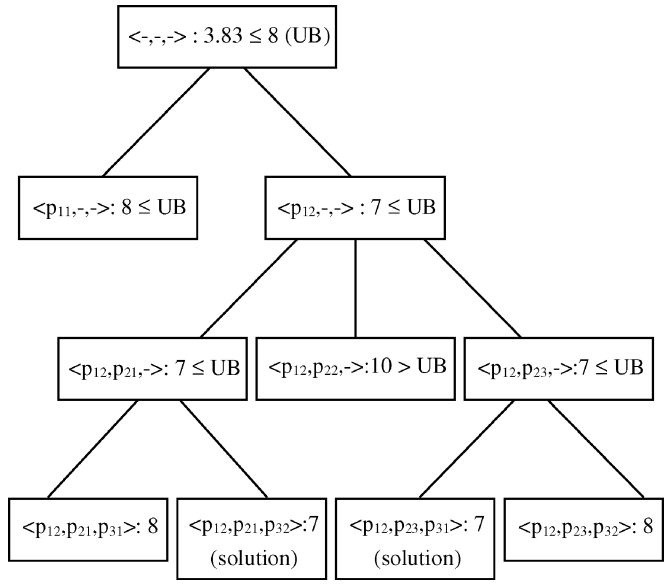

Fig. 5. A* heuristic search tree.

IV. GA Solution to MQO

A. GA Review

One of the most popular metaheuristic techniques used for solving complex optimization problems is the genetic algorithm. Since its introduction in [17] by Holland, the GA has attracted a lot of interest from the computer science community, and it has been successfully used for developing efficient solutions to many NP-complete problems. Goldberg [18] illustrates the practicality of GA with a detailed survey of GA applications.

GA simulates the evolution concept of biology. This simulation includes probabilistic methods using the principles of evolution. In GA, the main data structure is a simple vector (called a *chromosome*) representing a solution instance of a problem. Elements of the chromosome (called *genes*) contain a part of the solution of the problem. The quality of the solution instance (i.e., a chromosome) is defined by its closeness to the optimal solution (called the *fitness* function).

GA iteratively searches for an optimal solution by using evolutionary operations (also called genetic operators). At the beginning, a pool of random chromosomes is generated representing a set of different solutions. Then, genetic operators are applied to the pool of chromosomes to generate new chromosomes for the next iteration. Two genetic operators used in GA are as follows.

- *Crossover Operation*: In the crossover operation, sections of the parent chromosomes are exchanged to create new offspring chromosomes.
- *Mutation Operation*: New chromosomes are also generated by randomly modifying a few genes of an existing chromosome.

A selection technique must be used to determine which chromosomes survive to the next generation. The simplest approach can be defined as giving the better chromosomes a higher chance to survive to the next generation. This is what actually happens in the evolution process. However, sometimes a mutation or crossover operation on an unfit chromosome may produce a very fit chromosome. Therefore, preserving the diversity is also important. The most commonly used selection techniques are as follows.

- *Roulette Wheel*: All chromosomes of a generation are placed on a roulette wheel such that the area of the section of the wheel

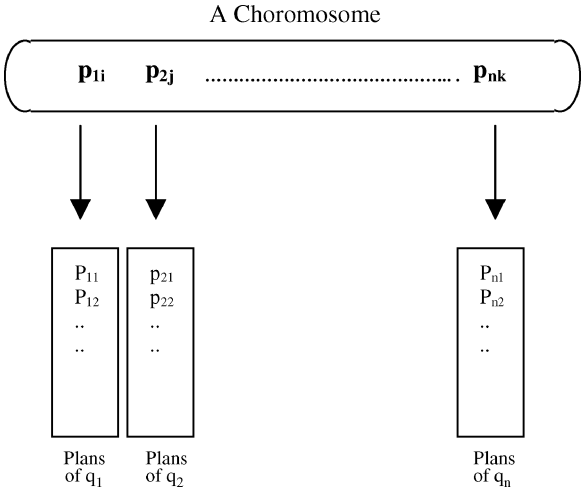

Fig. 6. Chromosome model of MQO.

corresponding to an individual chromosome is proportional to its fitness value. Repeatedly, random numbers are generated to represent points on the wheel, and chromosomes containing these points are picked for the next generation. This process is repeated as many times as the population size. Thus, it is possible to select the same chromosome several times.

- *Tournament*: Randomly, r (r is the tournament size, and it must be at least 2) chromosomes are picked from the population, and the chromosome with the best fitness value is chosen for the next generation from the r-element group. This process is repeated as many times as the population size of the next generation. Using this technique, too, it is possible to select the same chromosome several times.
- *Truncation (Elitism) Selection*: First, all the chromosomes are ranked from the best to the worst according to their fitness values. Then, n chromosomes are chosen among the top T chromosomes each with the same probability.

Through the iterations, GA explores the search space of the solutions of the problem. In this search, metaheuristics defined by genetic operators crossovers and mutations are used. In addition, the selection function determines which solutions survive. These three parameters determine the efficiency of the solution in terms of its speed and the solution quality.

B. GA Modeling of MQO

MQO problem can easily be modeled for a GA. In the context of MQO, a chromosome corresponds to a solution instance for the set of queries of the MQO problem. In a chromosome, each gene of a chromosome represents a plan to the corresponding query. Fig. 6 shows the structure of a chromosome. Each gene of a chromosome corresponds to a query. The value of the gene is the plan selected for the evaluation of the corresponding query.

To select the chromosomes for the next generation, the quality of the solution represented by the chromosome is used. This quality is represented by the fitness function, which is simply the inverse of the total execution time of all the tasks in the selected plans for the queries.

Under this modeling, MQO is also very suitable for genetic operations. Mutation and crossover operations can easily be defined to produce new valid solution instances.

Since a gene in a chromosome represents the plan selected for the query corresponding to the gene position, in the mutation operation we only replace the plan number with another randomly selected valid plan's number for that query. Therefore, a mutation operation always generates valid solutions.

Different crossover operations can be applied to chromosomes, such as one-point, segmented, and multipoint crossovers. In our representation scheme, all of these crossover techniques produce valid solutions for the MQO problem. If two chromosomes are representing two valid solutions of the same MQO problem, then, any crossover operation on these two chromosomes produces new chromosomes representing valid solutions for the same MQO problem. Regardless of the crossover type and positions, since all chromosome segments that are going to be exchanged to produce a new chromosome represent valid plans for their corresponding queries, the new chromosome obtained by appending these segments represent a valid solution of the MQO problem.

V. Experimental Results and Analysis

In this section, experimental results for comparisons of different GAs and their comparisons with the A-star algorithm are presented. Our experiments were performed on a PC with a 2.4-GHz Pentium IV processor. All of the algorithms were implemented in the C programming language. The following parameters have been used for generating different input sets:

1) number of queries (Q);
2) number of tasks in tasks pool (T);
3) number of plans in plans pool (P);
4) minimum and maximum and numbers of plans per query (PQm and PQM);
5) minimum and maximum numbers of tasks per plan (TPm and TPM);
6) minimum and maximum values for execution time of tasks (ETm and ETM).

The input generator generates the MQO problem instances by using the above parameters in the following order.

1) First, tasks are generated randomly by using two parameters, namely, the total number of tasks and minimum and maximum execution time values of tasks. Then, for each task generated, its execution time is selected as a value between minimum and maximum execution time values of the tasks.
2) After the tasks have been generated, the input generator decides the number of tasks by using the minimum and maximum number of tasks per plans parameters. Although many plans may share the same task, it is guaranteed that no two plans have exactly the same set of tasks.
3) Finally, queries are generated randomly by using the parameters defining the number of queries and minimum and maximum number of plans per query. Each query has its own distinct plan set, and therefore, a plan cannot solve more than one query. However, tasks can be shared by different queries since there can be plans solving different queries using the same task(s).

We use the minimum and the maximum numbers of tasks per plan and plans per query in generating random MQO instances. However, since mostly the averages of these two parameters (called as PQA and TPA, respectively), together with the number of queries, affect the size of the MQO problem instance, we use the following single parameter called *InputSize* to represent the MQO problem input size:

$$\text{InputSize} = Q \times \text{PQA} \times \text{TPA}.$$

Since InputSize provides a good estimation for the size of the search space that must be explored to solve the MQO problem, it can be used to estimate the execution time of heuristics solution technique exploring the search space. InputSize and total optimum solution value of an MQO problem are not related. Regardless of the number of plans

TABLE I
TYPES OF GAS

Genetic Algorithm	Selection Type	Crossover type
GA_1	Tournament	Segmented
GA_2	Roulette Wheel	Segmented
GA_3	Truncate	Segmented
GA_4	Tournament	Single Point
GA_5	Roulette Wheel	Single Point
GA_6	Truncate	Single Point
GA_7	Tournament	Multiple Point
GA_8	Roulette Wheel	Multiple Point
GA_9	Truncate	Multiple Point

per query, at the final solution, each query will have a single plan. Therefore, the optimum solution value is only related to the number of queries, number of tasks per plan, and execution times of the tasks.

To compare the heuristic algorithms developed for an NP-complete optimization problem, two features of these algorithms must be studied. The execution times of the algorithms and how close are the solutions obtained by these algorithms to the real optimal solution. Since the A* algorithm is guaranteed to produce an optimal solution, we only need to determine how close the GA solutions are to an optimal solution to be able to assess the success of the GA solution. We will use the term "solution time" to represent the total execution times of the tasks of the selected set of plans determined as the solution plan for the input MQO problem by the two MQO algorithms; namely, the GA and the A* algorithms.

A. Comparison of GAs

On the basis of the selection and crossover operation types, we have implemented nine different versions of GAs. Table I shows corresponding GA types with their features.

The following parameters are used in the implementation of GAs.

- *Population Size:* Total number of individuals (chromosomes) in each generation.
- *Maximum Number of Genes to Transfer:* Maximum length of the crossed segment in segmented crossover operation and maximum number of genes transferred in a multiple-point crossover operation.
- *Minimum Number of Genes to Transfer:* Minimum length of the crossed segment in segmented crossover operation and minimum number of genes transferred in a multiple-point crossover operation.
- *Tournament Size:* Number of individuals entering a tournament in tournament selection technique.
- *Truncate Ratio:* Ratio of the best individuals, which are sorted according to their fitness values, used for producing the next generation.
- *Mutation Ratio:* Probability of mutations in a single gene.

In all experiments, the values in Table II are taken as input parameters. These parameters are in the range of typical GA parameters [18]. For some of them, such as tournament size and mutation rate, we have tried a few different parameters and selected the ones giving the best results. All nine different GAs are tested with the same 25 random input cases. Solution times found by the GAs are given in Fig. 7. All

TABLE II
PARAMETER VALUES USED IN THE SIMULATION OF GAS

Population Size	100
Number of Generations	10
Maximum number of genes to transfer	5
Minimum number of genes to transfer	2
Tournament size	10
Truncate ratio	50 (%50)
Mutation ratio	1 (%1)

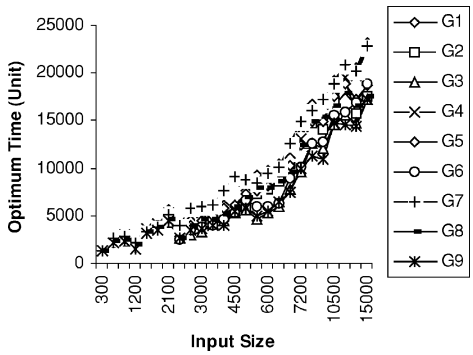

Fig. 7. Solution times of GA.

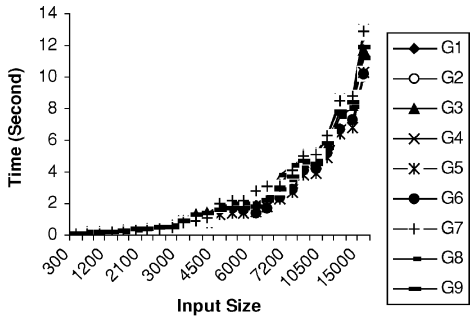

Fig. 8. Execution times of GA.

nine different versions of GAs have similar behaviors. As it is seen from Fig. 7, GA_3 and GA_9 give the best result. Both GA_3 and GA_9 use a truncate selection mechanism and a non-single-point crossover operation. There are also some sudden drops for all algorithms even though in general the solution times are increasing with the increasing InputSize parameter. Since InputSize is a rough estimation of the search space of the problem and solutions depend on the number of queries and the maximum number of tasks per plan, such drops are possible.

The execution times of the GAs are given in Fig. 8. The differences among the execution times of GAs are negligible. Tournament selection is the fastest among the selection techniques since it does not require large sort operations. Therefore, the GA technique using the tournament selection has the smallest execution time. On the other hand, roulette wheel is the slowest one because it includes sorting and construction of roulette wheel from the sorted individuals. Truncate selection includes only sorting.

Since the execution times of different GAs do not vary much, solution time values obtained by the GAs can be used to rank these GAs.

TABLE III
INPUTSIZE PARAMETER VALUES USED IN THE EXPERIMENTS

Exp. Set	Input Size	Number of Queries	Number of Plans per Query	Number of Tasks per Plan
Set 1	**250→ 750**	**5→ 15**	[4,6]	[9,11]
Set 2	**250→ 750**	5	**[4,6] → [14,16]**	[9,11]
Set 3	**250→ 750**	5	[4,6]	**[9,11] → [29,31]**

Although there are not big differences among the different versions of GAs for small input sizes, as the input size increases some GAs produce better solution time values. In that sense, GA_3 and GA_9 can be described as the better techniques.

B. Comparison of GAs With A-Star

The best GAs are GA_3 and GA_9. Since there is no significant difference between two best GAs (both of them are using truncate selection with nonsingle point crossover operation), we have picked one of them, namely GA_9, to compare the GA solution with A* solution.

To analyze the behaviors of GA and A* techniques with respect to the factors affecting the InputSize, we have performed three sets of experiments. In each set, to obtain different-sized problem instances, two of the three parameters affecting InputSize are fixed and the third one is varied. Table III summarizes all three experiment sets' parameters. Since two of the parameters affecting InputSize are averages of some ranges, we have also fixed the ranges. For both of these parameters, the range is chosen as 3 (i.e., it is like $[(n-1),(n+1)]$). In all three experiments, we have generated 11 different test cases representing MQO problem instances with input sizes from 250 to 750 (that is, problems with InputSize values 250, 300, . . . , 750).

In the first experiment set, the average number of plans per query and the average number of tasks per plan are fixed, former to 5 and the latter to 10. The number of queries is varying from 5 to 15 to generate desired problem sizes mentioned above. In the second experiment set, the number of queries and the average number of tasks per plan are fixed. The number of queries is fixed as 5, and the average number of tasks per plan is fixed as 10. The average number of plans per query is varying from 5 to 15. Finally, in the third experiment set, the number of queries and the average number of plans per query are fixed, both to 5, and the average number of tasks per plan is varying from 10 to 30.

In all the three sets of experiments, we have obtained both the solution times of the MQO problems generated by the GA and A* algorithm and the execution times of these two algorithms. Each experiment is repeated for ten randomly generated test cases, and averages of these tests are used in the graphics that demonstrate the results of both algorithms. Since the A* algorithm is guaranteed to produce the optimal (minimum) execution time for any MQO problem instance, by comparing the solutions of the two algorithms, we basically want to determine how close the solution generated by the GA is to the optimal solution. In addition, by comparing the execution times of the two algorithms, we want to determine how much more GA is efficient than the A* algorithm.

Solution times obtained for the two algorithms are given in Figs. 9–11 for the three different experiment sets explained above. As it is seen from these figures, for all the inputs, both GA and A* produce almost the same results. However, as the input size gets larger,

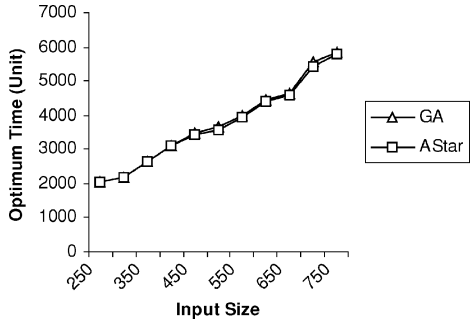

Fig. 9. Solution times corresponding to the first set of parameters.

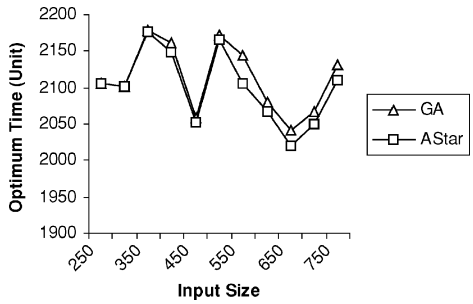

Fig. 10. Solution times corresponding to the second set of parameters.

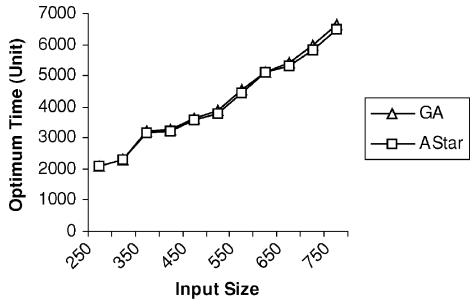

Fig. 11. Solution times corresponding to the third set of parameters.

since the execution time of A* increases exponentially, we could not obtain any result for it in a reasonable time. These results show that the GA solution is almost as good as the real optimal solution obtained by the A* algorithm. That is, in most cases, both solutions are exactly the same, and only in a few cases they differ. Also, when they differ, the difference between two solutions is less than 1%.

Figs. 12–14 show the execution times of both the algorithms for the three different input sets. As it can be seen from these figures, the A* algorithm can be used only for small input sizes.

As a summary, we can easily say that a GA produces almost the same result with A* in much less time, especially when the input size of the problem gets larger. GAs produce high-quality solutions (either optimal or very close to optimal) in a remarkable time. It takes too much time for an A* algorithm to execute for large inputs since its execution time (and memory requirements) grows exponentially. Therefore, for large inputs, it is better to use a GA for the MQO problem. We have carried out experiments with much larger input sizes with GA. Since the execution time of a GA grows linearly with the problem size, it is possible to solve much larger problems with it. The number of

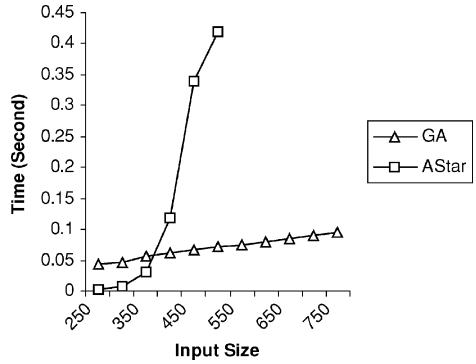

Fig. 12. Execution times corresponding to the first set of parameters.

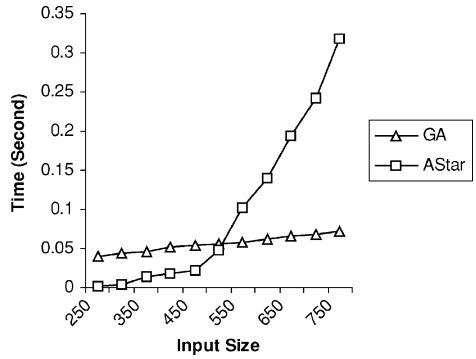

Fig. 13. Execution times corresponding to the second set of parameters.

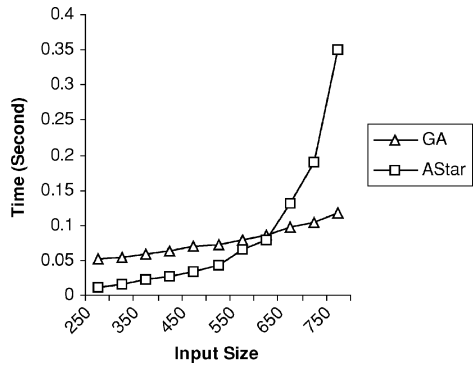

Fig. 14. Execution times corresponding to the third set of parameters.

generations in a GA is chosen as ten for the comparisons of the GA and A* algorithm, which is quite small for most GA applications. We have also made tests with a larger number of generations. Although the execution time of a GA depends on the number of generations linearly, as the input size of the problem gets larger, the execution time of the GA will be much more smaller than the execution time of the A* algorithm. Furthermore, for the problem sizes presented in Fig. 9–13, with 20 generations, almost for all inputs GA obtains the optimal solutions.

For the MQO problem, the number of queries affects the performance of the algorithms much more than the number of plans per query or the number of tasks per plan. Since increasing the number of queries increases the depth of the search tree, it has the most significant effect on the execution time. However, increasing the number of tasks per plan affects the number of children in a node in the search tree. This is also a very important factor on the execution time; but in general, its effect is usually less than the effect of the number of queries. Finally, increasing the number of tasks per plan directly affects the computation time related to a node in the search tree. Therefore, this parameter has a linear effect on the execution time cost.

VI. Conclusion

This correspondence presents evolutionary techniques for solving the MQO problem. Since MQO is an NP-hard problem, previously some heuristics solutions have been developed for it. Successful A* heuristics have been used for finding optimal solutions to the moderately sized (up to ten queries) MQO problems. In addition, rather than trying to obtain an optimal solution, some work has focused on finding near-optimal solutions in less time. GA has proven to be a very good candidate for such an approach, since with the GA technique described in this correspondence, very large instances of the MQO problem have been shown to be solvable very efficiently. The solutions obtained with the GA technique have also been shown to be very close to the solutions obtained by the A* heuristics.

References

[1] E. F. Codd, "Relational completeness of data base sublanguages," in *Data Base Systems*, R. J. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[2] U. S. Chakravarthy and A. Rosenthal, "Anatomy of a modular multiple query optimizer," in *Proc. VLDB*, 1988, pp. 230–239.

[3] S. Chakravarthy, "Divide and conquer: A basis for augmenting a conventional query optimizer with multiple query-processing capabilities," in *Proc. 7th Int. Conf. Data Eng.*, Kobe, Japan, Apr. 8–12, 1991, pp. 482–490.

[4] A. Cosar, "Design and experimental evaluation of a multiple query optimizer," Ph.D. dissertation, Comput. Sci. Dept., Univ. Minnesota, Minneapolis, 1996.

[5] M. Astrahan *et al.*, "System R: A relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, no. 2, pp. 97–137, 1976.

[6] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 3rd ed., Reading, MA: Addison-Wesley,1999.

[7] T. Sellis, "Multiple query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[8] P. Kalnis and D. Papadias, "Multi-query optimization for on-line analytical processing," *Inf. Syst.*, vol. 28, pp. 457–473, 2003.

[9] A. Cosar, E. P. Lim, and J. Srivastava, "Multiple query optimization with depth-first branch-and-bound and dynamic query ordering," in *Proc. CIKM 93*, 1993, pp. 433–438.

[10] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Dallas, TX, 2000, pp. 249–260.

[11] K. Shim, T. Sellis, and D. Nau, "Improvements on a heuristic algorithm for multiple-query optimization," *Data Knowl. Eng.*, vol. 12, no. 2, pp. 197–222, 1994.

[12] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan, "Pipelining in multi-query optimization," in *Proc. 20th PODS*, Santa Barbara, CA, 2001, pp. 59–70.

[13] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, "Materialized view selection and maintenance using multi-query optimization," presented at the SIGMOD Conf., Santa Barbara, CA, 2001.

[14] A. Y. Halevy, "Answering queries using views: A survey," *VLDB J.*, vol. 10, no. 4, pp. 270–294, 2001.

[15] C. Liu and A. Ursu, "A framework for global optimization of aggregate queries," in *Proc. (CIKM)*, 1997, pp. 262–269.

[16] U. Herzog and J. Schlosser, "Global optimization and parallelization of integrity constraint checks," in *Int. Conf. Manage. Data*, 1995, pp. 186–205.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. Michigan Press, 1975.

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