JOURNAL OF INFORMATION TECHNOLOGY AND ENGINEERING
Vol.3 No.1 Jan-June 2012 pp. 44-51
ISSN: 2229-7421

www.ispjournals.com

Query Optimization by Genetic Algorithm

Dr. P.K.Butey[1], Prof. Shweta Meshram[2] & Dr. R.L. Sonolikar[3]
[1]Kamala Nehru Mahavidhyalay, Nagpur.
[2,3]Priyadarshini Institute of Engineering and Technology, Nagpur.

Abstract: Genetic Algorithms are a powerful search technique based on the mechanics of natural selection and natural genetics that are used successfully to solve problems in many different disciplines.
In this paper we present genetic algorithms in one of the most important optimization problems in computer science, database query optimization for large join query.
Current query optimization techniques are inadequate to support some of the emerging database application. In this paper, we outline a database query optimization problem and describe the adaptation of genetic algorithm. And comparison between simple sql queries having five join and same query using genetic approach. And also give basic overview of the Carquinyoli Genetic Optimizer based on Genetic Programming.

Keywords: Genetic Algorithms, Query Optimization

1. INTRODUCTION

Genetic algorithm is becoming a widely used and accepted method for very difficult optimization problem. It has been used to solve a wide range of problem such as optimization, data mining, games, evolving behavior in biological communities etc.

Users really need to use very large join queries to support them in their business decisions [1]. Moreover, the complexity of these queries would increase if the DBMSs could handle them easily. With these common scenarios, current commercial DBMSs are becoming unable to return satisfactory results preserving the minimum performance requirements [2]. Specifically, dynamic programming techniques applied to query optimization, present both time and memory limitations. Since the search space grows exponentially with the linear increase of the number of relations involved in the query, these algorithms need to save an exponentially increasing number of partial plans into memory. This process is very time consuming and usually ends without a solution when the optimizer runs out of memory. Different approaches have been proposed to remedy this situation:

This paper is organized as follows. Section 2 contains introductory material providing some general working principles of Genetic Algorithm. Section 3 deals with different optimization algorithm for optimize large join query problem, with comparison in terms of estimation running time between simple sql query having five join and same query using genetic approach. And section 4 deals with basic overview of The Carquinyoli Genetic Optimizer based on Genetic Programming.

2. GENETIC PROGRAMMING IN QUERY OPTIMIZATION

In Genetic programming (GP) The basic idea is to obtain a best solution, called program originally, to solve a problem using evolutionary methods [3,4].
The basic behavior of this type of algorithms is as follows.

An initial set of programs is created from scratch. In this paper we represent programmed as tree structures, since they are the most suitable approach given that QEPs in DBMSs are usually tree-shaped. This set is also called the initial population.

Once the initial population is created, we iteratively apply a set of genetic transformations on the members in the population. The primary transformation operators are crossover and mutation.

The former works by changing two (or more) programs (or tree structures) combining them in some manner; the latter by modifying a single tree structure.

Each iteration of the algorithm is called a generation. At the end of each generation, a third genetic operation called selection is applied in order to eliminate the worst fitted members in the population. After applying these operations the algorithm obtains the next generation of members.

A stop condition ensures that the algorithm terminates. Once the stop criterion is met, we take the best solution from the final population. One of the typical applications for this type of algorithm is to solve optimal path search problems. In these problems, each member in the population represents a path to achieve a specific objective and has an associated cost.

Query optimization can be reduced to a search problem where the DBMS needs to find the optimum QEP in a vast search space. Each execution plan can be considered as a possible solution program for the problem of finding a good access path to retrieve the required data. Therefore, in a genetic optimizer, every member in the population is a valid execution plan. Intuitively, as the population evolves, the average plan cost of the members decreases [3, 5].

3. THE LARGE JOIN QUERY PROBLEM

The significant growth of the amount of data required in order to make the right decisions in nowadays businesses makes current query optimizers inadequate in some situations. Businesses providing banking services are a good example of the typical customer that needs very large storage capacity and very large and complex database designs. In such users really need to use very large join queries to support them in their business decisions. Different approaches have been proposed to remedy this situation [3, 5, 6]

Heuristic algorithms: the search space is reduced using estimates. They are usually very fast, but rarely find the optimum solution

Random algorithms: a random walk across the search space is performed in order to find a near optimum solution. Different policies lead to different algorithms, namely Iterative Improvement, Simulated Annealing, hybrid algorithms, etc

Genetic algorithms: inspired in natural selection, genetic algorithms try to find an optimum among a population. This population suffers from constant transformations, performed using three major types of operation: selection, combination and mutation.

Genetic Programming and Query Optimization

Query optimization can be reduced to a search problem where the DBMS needs to find the optimum query execution plan (QEP) in a vast search space. Each QEP can be considered as a possible solution (or program) for the problem of finding a good access path to retrieve the data required by a query. Therefore, in a genetic query optimizer, every member of the population is a QEP. Optimizing queries involving a large number of joins using genetic algorithms was introduced by Bennet el al. and tested later by Steinbrunn et al. showing that it is a very competitive approach [5, 7].

Genetic Algorithm with sql join query having 5 relations.

Considering the relational tables as follows

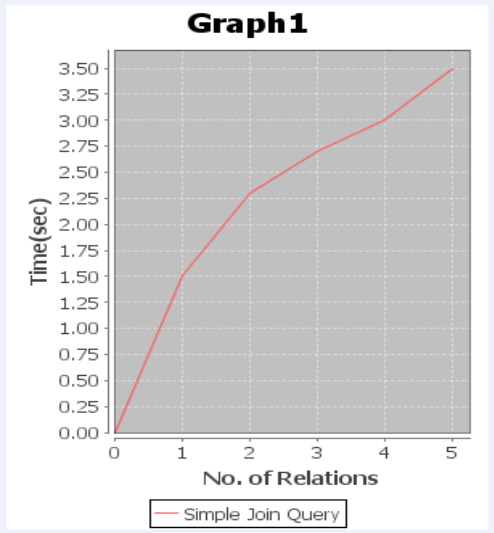
- Person, Orders, Product, Department, and Quantity with suitable dataset init.Following is the Simple Join Query to fetch the data from it on given certain conditions.

```
select
p.LastName, p.FirstName, o.OrderNo, o.p_id, pd.productname, pd.prd_id, s.sales_id, dept.departname, qty.quantity
from Persone p
inner join Orders o on p.p_id = o.p_id
inner join Product pd on o.prd_id = pd.prd_id
```

```
inner join Sales s on s.prd_id = pd.prd_id,
inner join Department dpt on s.sales_id = dpt.sales_id
inner join Quentity qty on dpt.dept_id = qty.dept_id
and qty.prd_id = qty.prd_id
order by p.LastName
group by dpt.dept_id;
```

By running the simple join query in SQL Server 2005 with

Produces the graph showing the time required to execute the query.



Applying Genetic Algorithm:

Now apply the (Sequential) Genetic Algorithm on the above simple join query with 5 join relations

Procedure:

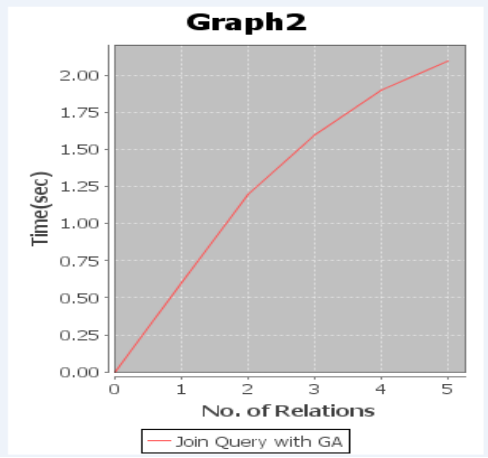
Here we consider the above query with N relations where (N=5)

1. Working principle of the Genetic Algorithm is to create the population (solution space).
2. After creating the population set the No. Generations and the No. Of Offspring which creates the basis for population
3. Working principle of the Genetic Algorithm is to create the population (solution space).
4. After creating the population set the No. Generations and the No. Of Offspring which creates the basis for population.
5. Then by taking the loop for all relations in the query creates the solution space i.e population by randomly selecting the relations using the rand (). Here for ith relation to be joined with randomly selected another relation considers the Left Deep tree [8].
6. After that as per the principle of GA select the parents from population and consider their chromosomes.
7. Calculate the fitness of each chromosome and select the best fit chromosome
8. For calculating the fitness the function to be taken as f(x) = x2.
9. Once the fitness is calculates for the chromosomes these chromosomes crossover with each other.
10. After completing the crossover mutation is takes place.

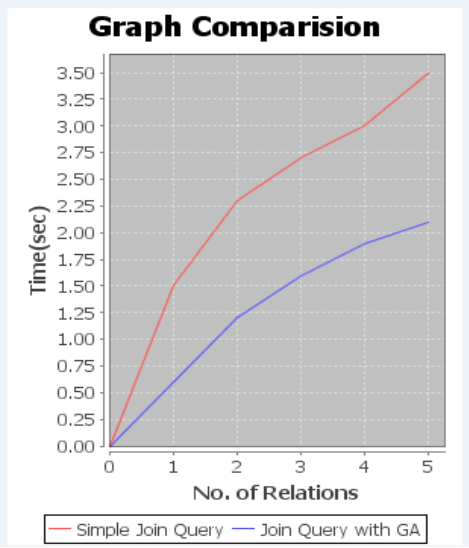
All the above steps are implemented in the SQL stored procedure.

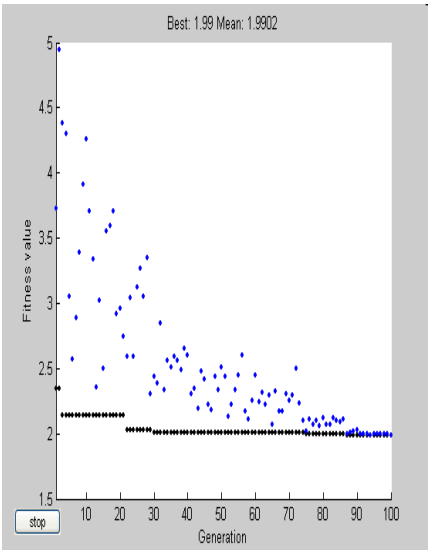
Note. For the following procedure create the table 'results' to store the result of computation of GA upon which the QEP runs and the graph to be calculate. Create procedure GeneticAlgowithjoins

After executing the above procedure in Sql Server 2005 the time taken for QEP of above sql query with 5 joins is shown in following graph.



Graph Comparison of time required for execution of the simple query before and after application of GA





Above graph shows the application of GA

1. Fitness values show the health of chromosomes selected in each Generation.
2. Graph shows the Best fitness value as 1.99 at 100th Generation.
3. In above graph the points at the bottom (black colored points) of the plot denote the best fitness values, while the points above them (blue colored) denote the averages of the fitness values in each generation. The plot also displays the best and mean values in the current generation numerically at the top.

4. The above graph displays the best fitness at each generation, shows little progress in lowering the fitness value and the average distance between individuals at each generation, which is a good measure of the diversity of a population.
5. Setting of initial range [1; 100], there is too little diversity for the algorithm to make progress.
6. The fitness value of an individual is the value of the fitness function for that individual. The best fitness value for a population is the smallest fitness value for any individual in the population.
7. In this case the fitness value of each individual (chromosome) is calculating with the function
1. $F(x) = x^2$
8. Where x is the health of the randomly selected individual (chromosome) at each generation.
9. At each generation the individuals are randomly selected and the fitness is calculated.
10. The individuals (chromosomes) with the best fitness values reproduce rapidly, taking over generations, and preventing the genetic algorithm from searching other areas of the solution space by skipping the weak fit chromosomes.
11. The number of individuals with the best fitness values in the current generation that are guaranteed to survive to the next generation. The genetic algorithm uses the individuals in the current generation to create the children that make up the next generation.
12. Increasing the population size enables the genetic algorithm to search more points and thereby obtain a better result.
13. The best fitness value improves rapidly in the early generations, when the individuals are farther from the optimum. The best fitness value improves more slowly in later generations, whose populations are closer to the optimal point.
14. Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. The selection function uses the scaled fitness values to select the parents of the next generation.

4. THE CARQUINYOLI GENETIC OPTIMIZER

We give a brief overview of CGO optimizers and proving that they can outperform the classical optimizers when the number of joins in a SQL query is large. The objects of study are those optimizers based on genetic programming.

In order to perform our analysis and present new ideas to improve the optimization of large join queries, we have implemented a new genetic optimizer based on genetic programming. Our optimizer is called the Carquinyoli Genetic Optimizer (CGO) [9] .

Basic Structures of CGO

We present the four basic structures used by CGO

The Database. CGO assumes to be working on a relational database schema. As usual, the schema contains a set of relations that are linked through primary and foreign keys (PK and FK, respectively). Each relation has an associated cardinality that corresponds to the number of tuples or records stored in that relation.

The Query. CGO assumes SQL to be the standard language used to express a query. However, in order to simplify the optimizer, CGO works with a reduced subset of the standard SQL language that allows for selections, projections, joins and sorting operations. The pseudo-SQL language used by CGO is called CGO-SQL.

The Query Execution Plan. The QEPs are the members of the population in a genetic optimizer. CGO assumes a QEP to be structured in a tree-shaped fashion.

The Population. The QEPs are organized in populations. Initially, a first population is created from scratch at random and, afterwards, the QEPs in the population are modified or eliminated.

1. CGO Operative Description

The optimization process starts with the creation of an initial set of programs (in our case, a program is a QEP), generally called members of the initial population by CGO. The initial population contains N members or programs. Each member in the population represents a way to achieve a specific objective and has an associated cost. In the case of the query optimization each QEP represents a way to solve the query introduced to the system. Starting with this initial population, usually created from scratch, two operations are used to produce new members in the population:

Crossover *operations,* which combine properties of the existing members in the population, and
Mutation *operations,* which introduce new properties into the population.

In order to keep the size of the population constant. CGO performs C crossover operations, choosing two random QEPs in the population each time, and mutation operations, choosing a single QEP, per generation.

After the application of the genetic operations, crossover operations have generated 2C new offspring and mutation operations have generated M new offspring. Thus, the population size has increased containing N + 2C + M members.

After this step, the cost for each new QEP in the population is calculated, and the QEPs are sorted by their cost. A third operation, usually referred as selection, is used to discard the worst fitted members, using this fitness function.

This process generates a new population containing N members, also called generation that includes both the old and the new members that have survived to the selection operation. This process is repeated iteratively for G generations until a stop condition ends the execution.

Every iteration returns a new population that is the evolution of the population returned in the previous iteration.

Once the stop criterion is met, the best solution is taken from the final population. Intuitively, as the population evolves, the average cost of the QEPs in the population decreases.

Algorithm 1, presents a simple description of the basic main procedure executed by CGO.

- First of all, the initial population is created from scratch (line 3).
- Once P is populated, we enter the main loop in line 4 and iterate for G generations.
- Every iteration in this loop represents the evolution and creation of a new generation of QEPs.
- For each generation, the crossover and mutation operations are applied (lines 5 and 6).
- Once all the operations are executed, the current population is merged with the new QEPs generated by the operations (line 7).
- And the selection operation is applied sorting the members in the population by cost and discarding the highest costed QEPs from the resulting population

Algorithm 1 CGO basic pseudo code

```
1: procedure CGO
main function
2: Population P, P1, P2, P3;
3: P = creatInitialPopulation ();
4: while (stop criterion is not met) do
5: P1 ← applyCrossoverOperations (P);
6: P2 ← applyMutationOperations (P);
7: P  ← P ∪ P1 ∪ P2;
8: P  ← applySelectionOperation (P);
9: end while
10: end procedure
```

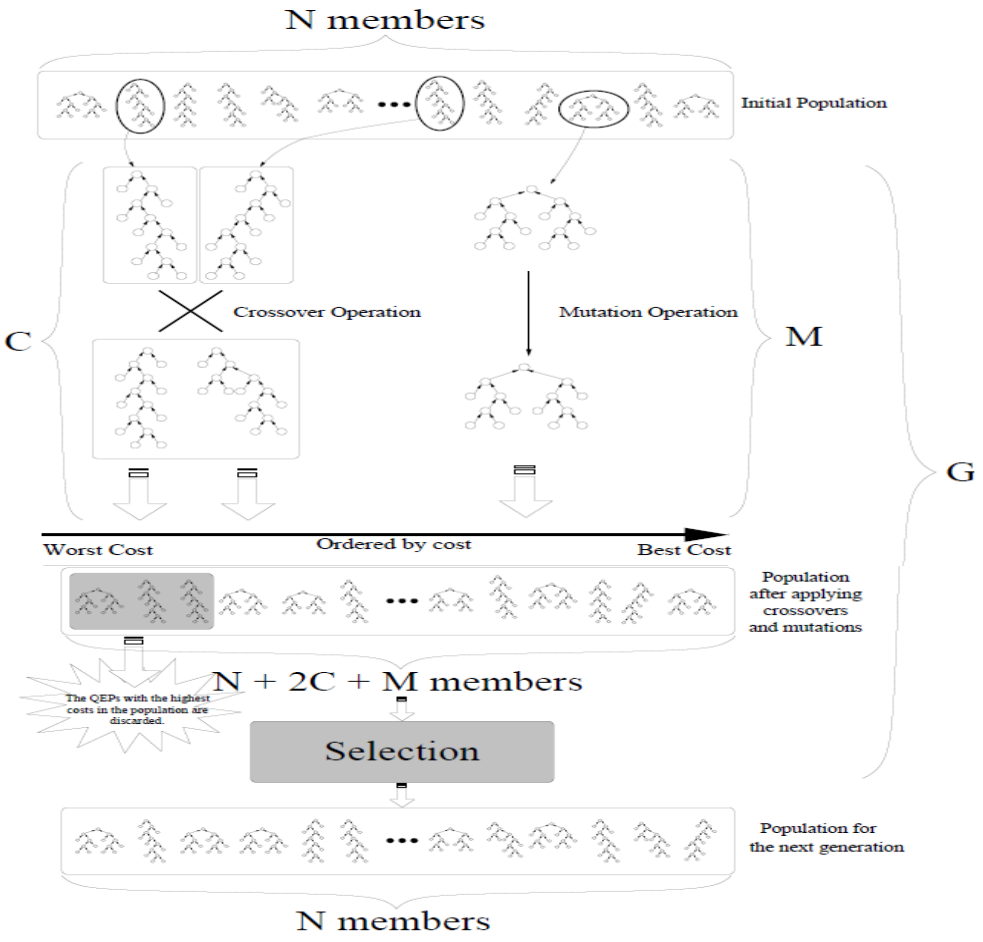


Fig 1. General view of the CGO behavior

CONCLUSION

This approach of GA is suitable for big databases which have the huge information and their goal is to perform the execution of large queries having more no of joins (more than 16 joins). The selection method and the best fitness function used for processing the chromosome (individuals) and the mutation process which decreases time and CPU cost according the no of relations.

The selection function assigns a higher probability of selection to individuals with higher scaled values. The range of the scaled values affects the performance of the genetic algorithm.

Lastly the graph which shows the best fitness value for chromosomes at every generation which can be used to study the behavior of function and can be useful to determine the function to select the fitness. This method can be use to optimize the join QEPs and time and cost required to execution. CGO optimizer based on genetic programming is capable to deal with large join query problem.

FUTURE SCOPE

Current query optimization techniques are inadequate to support some of the emerging database applications. Genetic algorithms however, are ideally suited to the processing, classification and control of complex queries for very-large and varied data.

This Approach of GA can be use to optimize the query having large no of joins (more than 16 joins to any no)

Again it can be use to minimize the memory requirement for QEPs.

References

[1] A. Swami and A. Gupta. Optimization of large join queries. In Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data, pages 8-17, Chicago, IL, June 1988.

[2] M. Jarke and J. Koch. Query optimization in database systems. ACM Computing Surveys, 16(2):111{152, June 1984.

[3] Melanie Mitchell, "An introduction to Genetic Algorithms", Prentice Hall of India, 2004

[4] Hsiung Sam, Matthews James, "An introduction to Genetic Algorithms", 2000 http://www.generation5.org/content/2000/ga.asp

[5] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In Proc. of the 1990 ACM-SIGMOD Conference on the Management of Data, pages 312{321, Atlantic City, NJ, May 1990.

[6] http://en.wikipedia.org/wiki/Category:Optimization_algorithms

[7] A. Swami and A. Gupta. Optimization of large join queries. In Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data, pages 8-17, Chicago, IL, June 1988

[8]http://www.deepdyve.com/lp/association-for-computing-machinery/left-deep-vs-bushy-trees-an-analysis-of-strategy-spaces-and-its-2GTEPmpJUv

[9] V. Munt¶es-Mulero, J. Aguilar-Saborit, C. Zuzarte, and J-L. Larriba-Pey. Cgo: a sound genetic optimizer for cyclic query graphs. In Proceedings of the International Conference on Computer Science