A Study of Execution Plan Aware Mutations for Genetic Cyclic Query Optimization

Victor Muntés Mulero, Josep Aguilar Saborit, Josep Lluis Larriba Pey y Calisto Zuzarte

I. INTRODUCTION

Several years of research have revealed powerful methods to optimize a query when the number of relations is small. Most of the commercial applications use variations of the same deterministic algorithm, carrying out an exhaustive search over the space of alternative strategies. This algorithm was first proposed for the System-R prototype DBMS [7].

The complexity of the optimization task depends directly on the number of relations involved in a query, i.e. the larger the number of relations the larger the search space of alternative execution plans. Queries in current applications usually involve a small number of relations, e.g., less than 10. Therefore, although the complexity grows exponentially with the number of relations involved in a query, it is still manageable.

A good survey on query optimization and other related issues can be found in [2]. However, current query optimization techniques are inadequate to support the needs of some of the newest database domains, such as artificial intelligence, human genome studies, etc. New applications on these domains are becoming more demanding on the number of relations involved in the query. As a consequence, the number of join operations increases drastically whereas the search space grows exponentially.

The application of genetic algorithms to large join query optimization was introduced in [1] and tested later in [8] showing that it is a good solution. As experimented in those and other papers, genetic algorithms can be used to find a near optimal solution to search problems in huge search spaces. In particular, query optimization can be reduced to an optimum search problem. Each execution plan is a solution to the problem of efficiently accessing data in a database with an associated cost.

In this paper, we propose new algorithms to implement mutation operations. These algorithms are more conscious of the execution plan structure. We have made sure that our mutation strategies always produce valid trees. We explain in detail how we implement these techniques in order to also deal with cyclic query graphs. Finally, we analyze the convergence of these methods, comparing the quality of the results found by these mutation techniques. We show that methods keeping previous information and more aware of the database optimization problem achieve better quality in the results and converge faster.

II. GENERAL DESCRIPTION

Commercial DBMS optimizers are not adapted to the needs of large join queries. Therefore, it is necessary to build an optimizer prototype to analyze the behavior of different genetic methods. The model of our prototype assumes that the access path chosen to retrieve data, given a query, is represented by an execution plan. Like in the classical representation, an execution plan will be described as a tree-shaped structure. The top node returns the desired results, and the leaf nodes represent scan operations on the relations involved in the query. Middle nodes are in charge of joining the data coming from lower operations.

As shown in Figure 1, we assume that there is a previous module that parses the query and transforms declarative statements into a graph. This graph contains a vertex for each referenced relation. Edges join a pair of vertices when a join condition exists between attributes of these relations appear in the query statement. We assume that there is only one join condition for each pair of relations.

There are two special cases that we consider separately:

- **Cross products**: cross products materialize joins between relations that are not directly connected on the query graph.
- **Artificial Joins**: artificial joins are join operations between two instances of the same relation.

In both cases, these join operations are very expen-
sive and increase enormously the size of the search space. Execution plans not containing any of these join types are referenced as valid tree as defined in [9].

A start population has to be created from scratch. In this study we create our model only takes into consideration valid trees.

As inequalities are rarely found in commercial queries, our optimizer prototype only considers equijoins in order to be simple and easily verified, is only based on I/O and does not take into account CPU cost.

Our genetic optimizer prototype uses the crossover operation proposed by Stillger and Spiliopoulou in [9] with some modifications, as our technique deals with acyclic and cyclic query graphs.

C. Mutation Operations

Crossover operations do not grant access to the whole search space, i.e. a population may not contain certain characteristics that might be necessary in the optimum execution plan. That makes necessary to introduce mutations to give an opportunity to add new characteristics that are not represented in any execution plan.

Mutation operations in previous proposals [1], [8], [9] focus basically on providing new execution plans. In this paper we show that the more aware the mutation of the structure of an execution plan the faster is the convergence of the algorithm.

We propose four different kinds of mutation:

- **Swap (S)**. A join operation is randomly selected and its input relations are swapped. As the algorithm can be used with cyclic graphs, only main joins will be chosen to perform the mutation.
- **Heuristic (H)**. An heuristic algorithm is used to transform an execution plan to a possibly better plan. In this paper, we present a very simple algorithm. The objective of the used heuristic is to ensure that the smaller relation is always in the build side of the hash join [6].
- **Random (R)**. A new random tree is introduced in that generation. Although this technique can-
not be considered strictly a mutation, it can be useful to introduce variance in the population.

- **Random Subtree (RS).** In order to keep the different properties of the execution plans in the population during the mutation operation, a subtree $S$ from a randomly chosen execution plan is selected. The remaining join operations, not included in $S$ are selected in random order until we have inserted all of them and, therefore, created a new complete execution plan.

By construction, these four mutations only produce valid execution trees, so, repairing actions are not needed.

### III. Analysis of results

In this section, several aspects of the behavior of mutations are studied. The steps of the analysis are:

- **Behavior of isolated mutations with acyclic graphs.**
- **Influence of the number of joins on mutations.**
- **Behavior of mutations combined with crossover operations with acyclic graphs.**
- **Analysis of the optimum number of mutations.**

Queries are generated randomly to provide more generic results. However, for each experiment, the same query is used to make comparisons between the different policies at hand.

Genetic algorithms are based on random techniques, therefore, two executions may differ in the quality of the yielded execution plan. For that reason, all our experiments are executed more than once. All the results presented in this section are based on averages on the figures obtained for every set of executions.

**Experiment 1. Behavior of isolated mutations with acyclic graphs.**

The first objective of this analysis is to determine the different characteristics of each mutation policy. The initial population is randomly created from an acyclic query graph with 21 relations and 20 join conditions. Mutations are used isolated, i.e. crossover operations are not used at all.

When a mutation policy is used alone we perform 60 mutation operations per generation. In the cases where policies are combined, we execute 30 operations of each type per generation. Table I summarizes the set of parameters used for this experiment.

Figure 2 shows the average cost of the best execution plan, after each generation.

**Swap (S).** As expected, this algorithm is limited by the quality of the initial population. We can assume that initially only half of the joins have their input relations in the right position (smaller table used as the build relation). Hence, as $S$ chooses randomly a join operation, by probability, at the beginning half of the mutation operations lead to better plans. As the population evolves, as shown in Figure 2, members in the population tend to be better, so the probability of generating a worse plan increases.

**Heuristic (H).** As we can see in Figure 2, after the first generations we have heuristically optimized the best execution plan in the initial population. The algorithm cannot find a better plan and it reaches an asymptotic behavior very soon. The speed of convergence and its speed in relation to $S$, depends basically on the fitness of the heuristic function. As it happened with $S$, heuristic is also bounded, as there is a set of characteristics that are never modified with this policy. For example, the order between joins is never altered. Therefore, this kind of mutation must rely on the **good** properties of the initial population.

**Random (R).** From the properties of this operation we can understand its performance. The more generations evolved, the worse chance to randomly find a **good** execution plan. With infinite time this algorithm would find the best solution. But its convergence is very slow and, as shown in Figure 2, 100 obtained are not enough to reach the quality of the results generated by $S$ or $H$.

**Random Subtree (RS).** The main difference between RS and $R$ is that RS keeps information of previous members in the population while $R$ just ignores any previous knowledge. However, in this experiment, RS performs worse than $R$. As we are only optimizing execution plans containing 20 joins, the search space is small enough to let $R$ find good solutions. Also, in those cases where the initial population members have expensive execution costs associated, $R$ may outperform RS because undesired properties are kept through generations.

Although the first two algorithms, $S$ and $H$, are
aware of the execution plan structures, they need from another operation to introduce new properties in the population. On the other hand, R and RS introduce new properties but lack of plan structure awareness. This is the main reason that leads us to combine these techniques.

**Random + Swap (R + S).** Although it is the weakest combination, it outperforms the four previous cases after a large number of generations (over 60 in this experiment). At the beginning its performance is very similar to the performance of S. Only after 30 generations, the new properties introduced by R show their effects allowing the algorithm to find better execution plans, beyond those obtained by H.

**Random + Heuristic (R + H).** As it happened between S and H, if the used heuristic function is better than a random function, R+H outperforms R+S.

**Random Subtree + Swap (RS + S).** Even taking into account that H always converges faster than S, RS+S converge faster than R+H. This situation is produced by the fact that RS keeps information of the previous members in the population. During the first 30 generations R+H converge faster, afterwards, the every time cheaper execution plans allow RS+S to preserve good information and achieve final execution plans twice cheaper.

**Random Subtree + Heuristic (RS + H).** Finally, the fastest convergence and best quality of results are obtained by the combination of RS and H. H provides fast convergence and RS provides new properties that allow the algorithm to be open to the whole search space, preserving previous information.

**Experiment 2. Influence of the number of joins on mutations**

Before we proceed with further experiments we need to test whether the number of joins has a significant influence on the observed behaviors for the different policies.

For that purpose we repeated the same test with 100 join queries. Table II shows the parameter set used for this experiment.

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_MEMBERS</td>
<td>250</td>
</tr>
<tr>
<td>NUM_REL</td>
<td>101</td>
</tr>
<tr>
<td>NUM_JOINS</td>
<td>100</td>
</tr>
<tr>
<td>NUM_GENERATIONS</td>
<td>150</td>
</tr>
<tr>
<td>NUM_CROSSOVERS</td>
<td>0</td>
</tr>
<tr>
<td>NUM_MUTATIONS</td>
<td>60</td>
</tr>
<tr>
<td>NUM_EXECUTIONS</td>
<td>10</td>
</tr>
</tbody>
</table>

As we can see in figure 3, our results show the same general trends:

- RS clearly outperforms R because the search space has been increased and the probability of finding a random low-cost plan decreases exponentially.
- Analogously, S also converges very slowly as it randomly swaps one join operation.
- S, R + S and RS + S do not reach a low-cost execution plan in a small number of generations.
- All policies using H show faster convergence.
- As in the previous experiment RS + H outperforms the rest of proposals.

We can conclude that the general behavior of the algorithms is the same as in the previous experiment, although differences between them become more noticeable.

Figure 4 shows the difference between the cost of the worst plan in a population after each generation and the best plan, for each mutation algorithm. After 30 generations, H has already collapsed all the members in the population to the same execution plan. So, it converges very fast although, as we have seen, it does not provide the best quality in results. In terms of convergence, the second fastest alternative is R + H followed by RS + H because of the difficulties of R to find a new low-cost plan compared to RS. Therefore, R + H converge faster although it gets worst results. The remaining algorithms, not using H, do not collapse in a unique execution plan due a lack of time.
Experiment 3. Behavior of mutations combined with crossover operations with acyclic graphs

Up to now, mutation operations have been isolated to better study their pure behavior. However, in a real system, genetic algorithms mix up selection and mutation operations with crossover operations. This experiment shows how the system evolves when we combine these mutation algorithms with crossover operations. Table III describes the parameters used for this experiment.

**TABLA III**

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_MEMBERS</td>
<td>100</td>
</tr>
<tr>
<td>NUM_REL</td>
<td>31 / 51</td>
</tr>
<tr>
<td>NUM_JOINS</td>
<td>30 / 50</td>
</tr>
<tr>
<td>NUM_GENERATIONS</td>
<td>100</td>
</tr>
<tr>
<td>NUM_CROSSOVERS</td>
<td>30</td>
</tr>
<tr>
<td>NUM_MUTATIONS</td>
<td>10</td>
</tr>
<tr>
<td>NUM_EXECUTIONS</td>
<td>5</td>
</tr>
</tbody>
</table>

Experiment 3 was first tested using an acyclic query graph involving 30 join operations. Figure 5 show differences with previous experiments:

- H performs almost the same as R + H and RS + H. This effect is produced by the variation introduced by crossover operations.
- The cost of execution plans produced by R + H and RS + H is still 25% lower than the cost of the plans yielded by H. The general trends, however, remain the same.

The same test was run on a 50-joins acyclic query graph using the best three mutation methods: H, R + H, RS + H. Figure 6 shows the results of this executions.

RS + H clearly outperforms the other two approaches, obtaining results 47.68% cheaper than H and 54.21% cheaper than those returned by R + H.

Figure 7 shows a similar convergence in the three cases.

Experiment 4. Analysis of the optimum number of mutations

Finally, in experiment 4, focuses in the analysis of performance varying the number of mutations. To achieve this objective, this experiment is designed following the parameters in Table IV.

**TABLA IV**

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_MEMBERS</td>
<td>400</td>
</tr>
<tr>
<td>NUM_REL</td>
<td>201</td>
</tr>
<tr>
<td>NUM_JOINS</td>
<td>200</td>
</tr>
<tr>
<td>NUM_GENERATIONS</td>
<td>400</td>
</tr>
<tr>
<td>NUM_CROSSOVERS</td>
<td>100</td>
</tr>
<tr>
<td>NUM_MUTATIONS</td>
<td>2/4/10/20/30/60</td>
</tr>
<tr>
<td>NUM_EXECUTIONS</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8, shows that the quality of final execution plans increases asymptotically when increasing the number of mutation operations performed by generation. As we can see in the plot, using over 30 mutations, in this case, does not provide a significant increase in the quality of results. In fact, results obtained with 4 mutations are very close to those obtained by 30 mutation. If we take into account that the number of mutation operations per genera-
tion increases the execution time of the optimizer, we may avoid using more than 10 mutation operations per generation.

The experiment also showed that the differences regarding the speed of convergence were not significant. Out of 200 generations, after 15 generations all the algorithms converged similarly.

![Cost of the final execution plan](image)

After generation

Cost (IO accesses)

A. Related Work

Good effort has been spent in obtaining execution plans for queries between 15 and 50 joins [11].

Other proposals have studied more general solutions even for those cases in which 50 to 100 joins are needed. The different proposed algorithms can be classified as Heuristic algorithms [8], Random algorithms [8], Genetic algorithms [1], [8], [9] and Hybrid algorithms [10].

As experimented in [8], genetic algorithms compete in quality of results and speed with the other approaches. Application of genetic algorithms to database query optimization was first proposed in [1]. The proposed algorithm is complex and expensive and may lead to invalid trees that have to be repaired. In addition, the execution plans generated lose almost all the structural information from their parents. [8] concludes that, besides these shortcomings, randomized and genetic algorithms are good solutions to optimize large join queries. In [9] genetic programming is used to build a new method to perform crossover operations. This method does not use chromosomes to represent execution plans, but the execution plans themselves. Only two mutation operations are used and execution plans produced by one of them may need to be repaired.

Moreover, [1] and [9] only deal with acyclic query graphs.

IV. Conclusions and Future Work

From the previous experiments we conclude that:

- Mutation operations must be conscious of the database structures and the execution tree constraints. At the same time, we show that they must provide variance. Therefore, we conclude that these two characteristics must be combined in order two obtain a good mutation operation, like Random-Subtree plus Heuristic.
- However, as crossover operations also introduce variance, database-aware mutation operations with little variance, like the Heuristic strategy, may perform similarly to those introducing more variance, like Random-Subtree plus Heuristic.
- Mutation operations must use information of execution plans from previous generations in order to converge faster to a low-cost solution.
- The number of joins does not affect significantly the behavior of the different techniques.
- Hybrid approaches (heuristic and genetic) yield good results and performance.
- Regarding the number of mutations per generation, there is a trade-off between a small number of mutations, that may not offer enough new possibilities in the search space, and a large number of them, that may yield results of the same quality as results yielded by executions with less mutations and, therefore, faster.

Currently, we are studying an efficient way to decide the optimum number of mutation operations to be performed per generation. Future work includes, extending this conclusions to systems supporting other join implementations and the use of indexes. Another objective is to study the implementation of other database-conscious mutation operations such as randomly introducing an interesting order.

**REFERENCES**


