An inside analysis of a genetic-programming based optimizer

Victor Muntés-Mulero,
Josep Aguilar-Saborit
and Josep-L. Larriba-Pey
DAMA-UPC
Computer Arch. Dept. Campus Nord. UPC
C/Jordi Girona, 1-3. Módul D6. 08034 Barcelona
Email: {vmuntes, jaguilar, larri}@ac.upc.edu
http://www.dama.upc.edu

Calisto Zuzarte
IBM Canada Ltd
DB2 IBM Toronto Lab.
8200 Warden Ave.
Markham, Ontario
Canada L6G1C7
Email: calisto@ca.ibm.com

Volker Markl
IBM Almaden Research Center
San Jose, CA, 95139
USA
Email: marklv@us.ibm.com

***Abstract*— The use of evolutionary algorithms has been proposed as a powerful random search strategy to solve the join order problem. Specifically, genetic programming used in query optimization has been proposed as an alternative to the limitations of dynamic programming with large join queries. However, very little is known about the impact and behavior of the genetic operations used in this type of algorithms.**

In this paper, we present an analysis that helps us to understand the effect of these operations during the optimization execution. Specifically, we study five different aspects: the age of the members in the population in terms of generations, the number of query execution plans (QEP) discarded without producing new offsprings, the average QEP life time in generations, the efficiency of the genetic operations and the evolution of the best cost.

All in all, our analysis allows us to understand the impact of crossovers compared to mutation operations and the dynamically changing effects of these operations.

I. Introduction

Determining the optimal join order is a well-known and important problem of query optimization in relational databases. Unfortunately, this problem is known to be NP-hard, and traditional query optimizers, which typically employ dynamic programming techniques [1], cannot handle such a large number of joins (usually over 15 joins) due to the exponential explosion of the search space. In addition, even in the event that the system has enough memory, finding the best fitted *query execution plan* (QEP) is tremendously time consuming. In these situations, optimizers either resort to heuristics [2] or fall back to greedy algorithms which do not consider the entire search space and may overlook the optimal plan.

The application of genetic programming to query optimization was proposed by Stillger et al. in [3], presenting a new class of crossover operation able to handle QEPs represented as tree structures. The Carquinyoli Genetic Optimizer (CGO) [4] further develops this idea extending the work to mutation operations and enabling CGO to allow for cyclic query graphs[1]. We use CGO since, to the best of our knowledge, it is the most complete and tested genetic programming-based optimizer. Also, CGO is coupled with the cost model used by the DB2 UDB optimizer. This is important to be sure that the conclusions extracted in this paper are based on a realistic cost function and, thus, that they can be generalized to any sound genetic optimizer.

We take a step further from the work in [3] and [4] by providing a means to understand the important aspects required to obtain a good QEP. Specifically, we study the effects of the different genetic operations in the internal evolution of the QEPs. Our analysis underlines the importance of crossover operations compared to mutation operations, although it also shows that mutation operations are essential to grant quality in the results, specially during the first generation of the optimization execution.

This paper is organized as follows. Section 2 introduces genetic optimization and CGO. Section 3 describes the genetic operations used in this work. In Section 4, we perform the analysis of the genetic operations. In sections 5 and 6, we present related work and conclude.

II. Genetic Programming in Query Optimization

Inspired by the principles of natural selection, genetic programming performs operations on the members of a given population, imitating the natural evolution through generations.

A genetic programming algorithm typically creates an initial population where each member represents a path to achieve a specific objective and has an associated cost. This first population is usually created at random 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

[1]We call *query graph* the graph extracted from the SQL statement, where each node is one of the relations accessed by the query and each edge represents one of the join conditions in the *where* clause. We say that a query graph containing N nodes is *cyclic* when the number of join conditions is larger than $N-1$, assuming that it is always possible to go from a node to any other node following the edges.

introduce new properties into the population. In order to keep the size of the population constant, a third operation, usually referred as *selection*, is used to discard the worst fitted members, using a fitness function. This process generates a new population, also called generation, that includes both the old and the new members that have survived to the selection operation. This is repeated iteratively until a *stop condition* ends the execution. Once the stop criterion is met, we take the best solution from the final population.

Query optimization can be reduced to a search problem where the DBMS needs to find the optimal QEP in a vast search space. Each QEP can be considered as a possible solution 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 QEP. Intuitively, as the population evolves, the average plan cost of the members decreases.

The Carquinyoli Genetic Optimizer (CGO) is presented in [4]. CGO assumes that a query execution plan (QEP) is a directed data flow graph, where leaf nodes represent the base access plans of the relations. Data flows from these nodes to the higher nodes in the graph. The non-leaf nodes process and combine the data from their input nodes using physical implementations of relational operations like SELECT, PROJECT or JOIN. The root node returns the final results of the query. CGO includes the basic QEP operations, which are typically used in most commercial DBMSs:

- Scan Operations: CGO allows for two basic scan operations: *sequential scan* and *index scan*. CGO considers a blocking technique in order to reduce the number of accesses to disk for sequential scans (in the DB2 UDB optimizer, this is known as *sequential prefetch* [5]). With the index scan, CGO takes into account some of the different index properties: the clustering factor of the data with respect to the index, the possibility of pushing down predicates, different methods for accessing leaf pages, prefetching techniques, etc.
- Join Operations: CGO allows for the three basic join implementations: *Hash Join*, *Nested-Loop Join* and *Merge-Scan Join*. For Hash Join CGO considers the use of bit filters [6] to reduce the number of internal I/O accesses.
- Other Operations: Besides scan and join operations, CGO allows for two more operations: Sorting and materializing (Temp Operator).

CGO also allows for the basic relational operations of projection and selection although these are implicitly included in the other operations. The attributes projected by each QEP operation are those attributes required for the *select* statement in the SQL query and those required by upper join conditions.

CGO is coupled to the cost function used by the DB2 UDB optimizer. Note that, although the conclusions presented in this paper can be generalized to any genetic optimizer, a highly inaccurate cost model could invalidate the study. Therefore, using the cost model of DB2, we can assure that the estimations of the costs of the QEPs are realistic.

CGO has been validated against the DB2 UDB optimizer, proving that, for large join queries, it can outperform the greedy algorithm used by the commercial DBMS when the memory resources are exhausted due to the size of the search space. Details on the comparison can be found in [4].

III. Genetic Operations

As introduced in Section II, genetic programming is based on three operations: crossovers, mutations and selection. In this subsection we focus on the specific aspects of these operations in CGO.

Crossover Operations. In order to combine different properties present in the members of a population, crossover operations randomly choose two QEPs of the population and produce two new trees preserving two subtrees from the parent plans. A detailed explanation of the implementation, the way it handles cyclic queries and a detailed example can be found in [7].

Mutation Operations. Due to the fact that crossover operations do only combine existing properties in the population but do not introduce new information, mutation operations are necessary to provide an opportunity to add new characteristics that are not represented in any of the QEPs in the population. Therefore, we must assure that any possible QEP in the search space is potentially reachable through the transformations performed by this mutations. Any QEP in the search space is defined by:

- the tree morphology (i.e. the shape of the tree)
- the join order in the QEP for a given shape
- the join methods used in the join operations
- the scan methods used in the scan operations.

With this purpose, CGO includes four different kinds of mutation:

- **Swap (S).** A join operation is randomly selected and its input relations are swaped. S is specially useful if we take into account that the cost of a join is not symmetric, i.e., it varies depending in which input is left and which is right. For instance, a hash join operation can drastically reduce the amount of I/O accesses depending on the size of the input relations. Specifically, in a hash join operation, one of the input relations is called *build relation* and it is used to create an in-memory hash table using the join attribute. The other input relation is called *probe relation* and its tuples are tested against the values in the hash table created before, in order to find a counterpart in the build relation. A hash join operation will incur in extra I/O when the hash table does not fit in memory. This algorithm is typically known as *hybrid hash join algorithm* [8]. In this case, it is usually advisable to use the smallest input relation as the build relation. Mutation S grants potential access to all tree morphologies and, therefore, it makes it possible to explore all the alternatives in asymmetric joins.
- **Change Scan (CS).** CGO randomly chooses a scan operation and changes the scan method if indexes are

available. This mutation grants access to all the possible scan methods for all the scan operations.

- **Change Join (CJ).** The optimizer randomly chooses a join operation and it changes its implementation to one of the other two available implementations. This operation makes it possible to implement any join operation using any of the three available join methods.
- **Random Subtree (RS).** 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, a new and complete QEP is created. This mutation grants the potential exploration of all the join operation orders, although it also allows to explore all the other dimensions.

By construction, these four mutations only produce valid execution trees, so, repairing actions are not needed. Therefore, we preserve the semantics of QEPs, avoiding computationally inefficient optimizers as suggested by De Jong in [9].

Selection Operations. An elitist algorithm is used in CGO to choose the best execution plans in each population, i.e. those execution plans with the lowest cost (computed using a subset of the cost model of the DB2 UDB optimizer), keeping the number of members in the population constant from one generation to another.

IV. Genetic Operations Analysis

This section aims at performing a comprehensive analysis of the genetic operations in a genetic programming based optimizer. Although the results are based on the operations of CGO, the results are general for any genetic optimizer which would use similar genetic operations. We report all the results analyzing the evolution through generations in order to let the conclusions be independent from the efficiency of the implementation in each possible optimizer. We study five basic aspects: the average age of the members in the population, the number of QEPs discarded without being used, the average QEP life time, the operation's efficiency and the best cost evolution. All the results are analyzed for the different genetic operations.

We executed the optimizer using queries involving 20, 30, 50 and 100 relations. We have randomly generated 25 different databases. For each database we have created 5 random queries and executed each one 4 times, performing 500 executions in total. We have not used known benchmarks like TPC-H since their schemas are very small an do not allow to build large join queries. In some cases, we have also performed extra executions with smaller queries, i.e. queries including 3 relations, in order to understand the differences between large join queries and small queries.

Due to a lack of space, we only present the results for 20 relations. However, the analysis in [7] shows that the general operation evolution trends do not depend on the number of relations involved in the query and, thus, the results presented in this paper can be generalized to larger queries.

For this experiment we use 200 members in the populations and 100 generations per execution. We analyze the case for an execution performing 26 crossover operations and 48 mutation operations (12 of each type) per generation, thus generating 100 new QEPs per iteration.

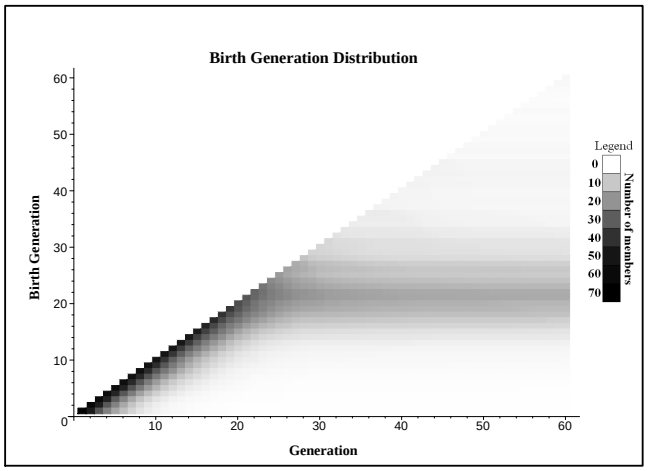

Fig. 1. Average birth generation distribution for each generation for queries accessing 3 relations.

We have tested other scenarios ranging from the application of only crossover operations to the application of only mutation operations. The general trend for each operation is the same across the different tested scenarios.

A. Population Ageing

Our first experiment analyzes the average birth generation for the members of the population in each generation. Our goal is to show that, after several generations, the probability of finding better QEPs than those present in the population decreases close to zero. We call this scenario *Dead Area*.

Figures 1 and 2 show an spectrum of the distribution of the average birth generation for very small queries (3 relations) and large join queries with 20 relations. Darker colors mean a larger concentration of members being born in the generation specified in the Y-axis. In general, we observe that, at the beginning, after a specific generation, most of the QEPs in the population where created in that generation, while QEPs created in previous generations are quickly discarded. After several iterations this trend changes, as observed in Figure 1. For the case of queries accessing 3 relations, after generation 20, most of the plans in the population are created between generations 15 and 25. That implies that later generations do not succeed in finding better plans to substitute old members in the population and we reach the *Dead Area*. In Figure 2 we observe that the *Dead Area* is not reached in the plots because after 100 generations we are still generating better plans that usually discard most of the plans created more than 10 generations ago.

It is interesting to note that the algorithm typically finds a near-optimal QEP several generations before the *Dead Area* is reached.

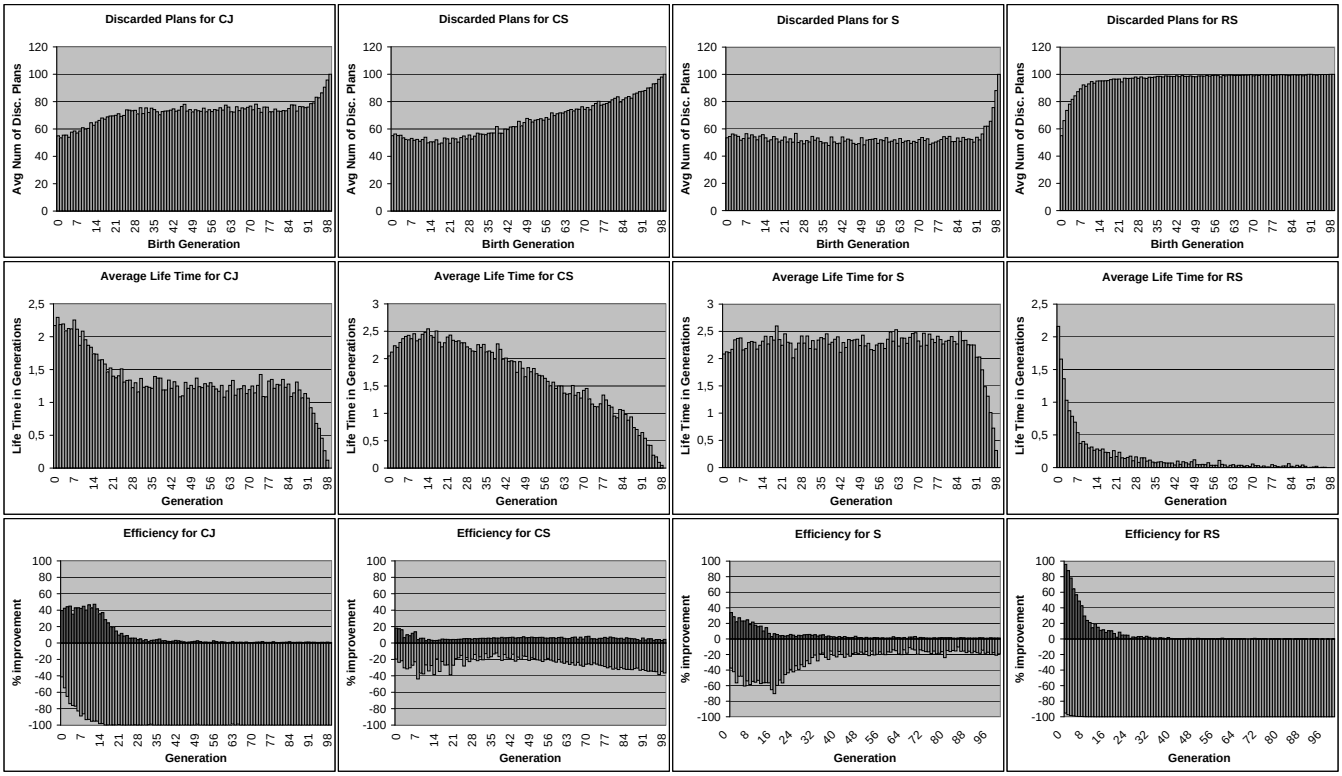

Fig. 3. Average operation behavior with 26 crossovers and 48 mutations per generation.

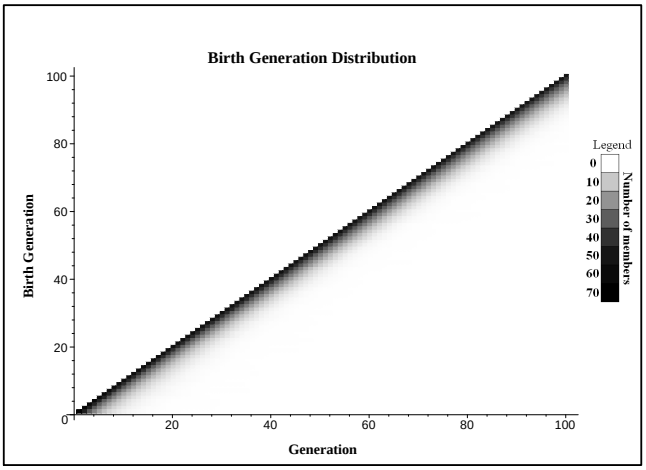

Fig. 2. Average birth generation distribution for each generation for queries accessing 20 relations.

B. Execution Plan Utilization.

The basic aim of a genetic operation is to introduce an interesting configuration in the population that helps the algorithm to take a step further towards a near-optimal execution plan. The combination of different genetic operations through several generations leads to the generation of increasingly improved plans. Therefore, the capability of a genetic operation to introduce a new property or configuration into the population is one of the measures to evaluate its efficiency.

In this subsection we analyze the number of plans that have been discarded without being used. That is, plans that are not used to generate new plans in a crossover or a mutation operation. These plans are very interesting in order to analyze the useless work carried out during the execution, since it takes time to generate them but the new properties introduced by them are never used.

The upper plots of Figure 3 show the average number of QEPs discarded without being used in a later operation for each mutation operation. It is important to notice that the values for the last 20 generations are skewed since we only examine the percentage of plans discarded without being used from the whole set of discarded QEPs. Of course, the probability of not using a discarded plan created in generation 99 is 1 since it dies in the same generation when it is created and, naturally, without being used. We have monitored all the executions and, in all the cases, the birth generation for all the plans alive in generation 99 was over generation number 77.

The results show that mutation operation RS is doing a lot of useless work since, in average, more than the 80 % of the QEPs generated by this operation are discarded without being used in a later operation after generation 5. On the other hand, almost 50 % of the QEPs generated by S are used in later operations, meaning that changes introduced by swap have a higher probability to have an impact in the genetic evolution. CJ and CS gradually loose potential as the average cost of the members in the population is lower.

Figure 4 shows the results for the crossover operation. The percentage of plans discarded without being used in other

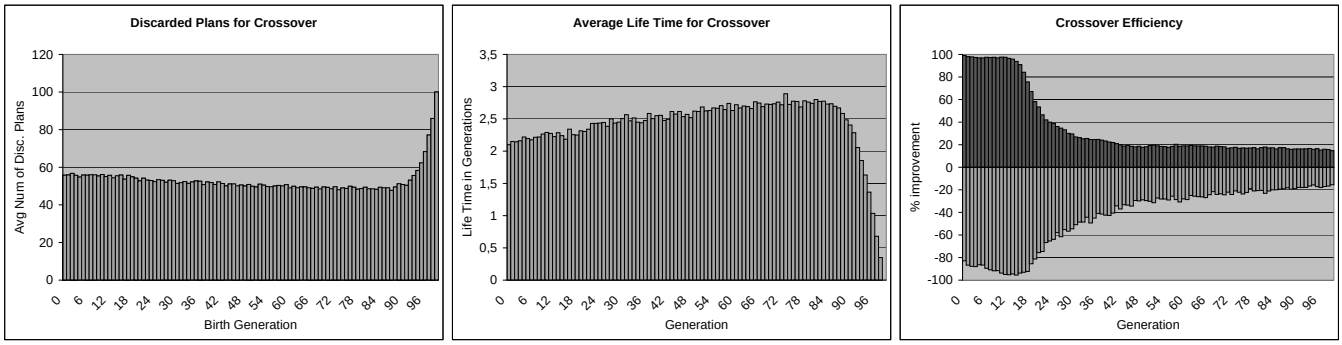

Fig. 4. Average crossover operation behavior

operations is always around 50 % which means that plans created after crossing two plans in the population result in an acceptable plan in half of the cases.

C. Execution Plan Life Time.

Another way to measure the effect of a specific genetic operation is to analyze the life time of the execution plans generated by that operation. Longer life times imply a higher probability for a given execution plan to be used in the next generations. In this subsection we analyze the life time of the execution plans grouped by the generation in which they were created.

The second row in Figure 3 gives the information on the average life time for each mutation operation. Again, the average life time for plans generated by RS show that after a reduced number of iterations, combined with other operations, RS basically carries out unnecessary operations since most of the plans die as soon as they are created. In constrast, the average life time of QEPs generated by CS decreases in an almost linear fashion. Also, the average life time of QEPs generated by CJ decreases linearly although, after 20 generations, it stabilizes to be rather uniform between 1 and 1.5 generations up to generation number 90, and then decays for the reasons explained before.

Figure 4 shows an increasing life time for QEPs created by crossover operations. Since this kind of operation does not introduce new information, but just combine existing information, it does not yield very high-costed QEPs. Also, taking into account that the mutation operations are usually less conservative and its probability to generate a low-costed QEP decreases, the probability to survive during more generations for QEPs obtained from crossover operations increases.

D. Genetic Operations Efficiency.

However, although the previous analysis provides us with an approximate picture of the behavior of genetic operations, it does not directly reveal the efficiency of the transformations introduced by those operations. For instance, an execution plan p_i, generated by the application of a specific operation to a parent execution plan p_p, could have a long life time thanks to an operation previously applied to p_p, which decreased the cost assigned to p_p below the average cost in the current population. Then, although the life time of p_i is long, this does not imply the efficiency of the last applied genetic operation. For this reason, in this subsection we directly study the efficiency of the operations by immediately calculating the percentage of improvement of the plan costs after the application of the genetic operation.

Formula (1) is used for mutation operations to calculate the average percentage of maximum improvement and worsening. p_c is the child plan and p_p the parent plan. For crossover operations we use (2) and (3) where p_{p1} and p_{p2} are the parent plans. The idea behind the equation for the crossover operation is to calculate whether the new plan is better than the average cost between both parent plans.

$$\%_M = \begin{cases} \frac{cost(p_c)}{cost(p_p)} \cdot 100 & \text{if } cost(p_c) \leq cost(p_p) \\ -\frac{cost(p_p)}{cost(p_c)} \cdot 100 & \text{if } cost(p_c) > cost(p_p) \end{cases} \quad (1)$$

$$r_{imp} = \frac{2 \cdot cost(p_c)}{cost(p_{p1}) + cost(p_{p2})} \quad (2)$$

$$\%_C = \begin{cases} (1 - r_{imp}) \cdot 100 & \text{if } r_{imp} \leq 1 \\ (\frac{1}{r_{imp}} - 1) \cdot 100 & \text{if } r_{imp} > 1 \end{cases} \quad (3)$$

The results for mutation operations are presented in the third row of Figure 3. We can observe that, for all the different mutation policies, the average maximum efficiency quickly decreases during the first 15 generations. Afterwards, CJ tends to generate only worse QEPs or QEPs with the same cost as the parent QEP. Curiously, although the worsening produced by CJ is very high from the beginning the average life time of its QEPs, as shown in the previous subsection, is kept over 1 during the optimization. This phenomenon is produced because in many cases CJ produces plans with very similar associated costs, for instance, in those scenarios were *Hash Join* and *Merge-Scan Join* have a similar performance impact. S is affected by the same phenomenon since, for example, swapping the input relations of a *Merge-Scan Join* does not have any impact regarding the cost associated to the parent QEP in our cost model. Therefore, although in average it does not improve the QEPs in the population, it does not worsen them either. CS achieves slightly better partial cost improvements, although they are still very marginal. RS just reacts as expected, generating low-cost QEPs at the beginning,

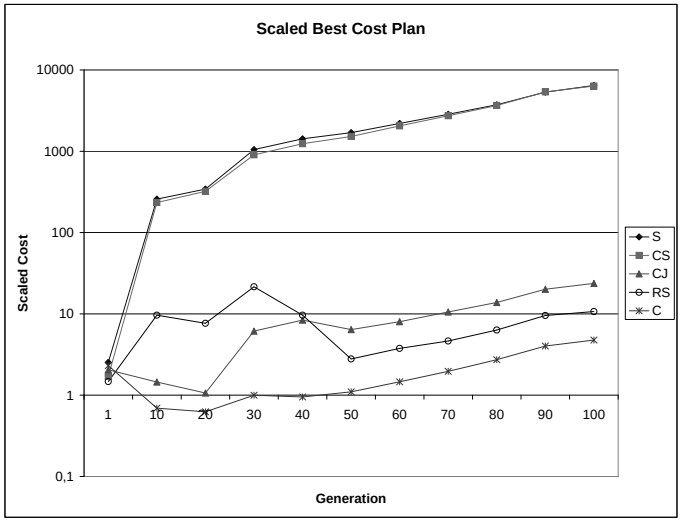

Fig. 5. Scaled best cost evolution compared to the combination of all the genetic operations.

but immediately decreasing its efficiency as the population is improved during the first generations.

Figure 4 corroborates our hypothesis about the conservative behavior of crossover operations. As the generations pass by, the gain and worsening obtained by this kind of operation decreases.

E. Best Cost Evolution.

Finally, we analyze the cost evolution for the best QEP in each generation. This analysis completes our study by showing the contribution of each genetic operation in isolation from the effects of the other operations compared to their combined execution. Namely, CGO optimizes each query using 6 different policies. The first five policies consist in applying only one type of mutation or crossover operation during the whole optimization. The last policy combines all the genetic operations.

We perform 150 executions involving the random creation of 5 different databases, creating a random query for each one and executing the 6 policies 5 times with queries involving 20 relations.

Figure 5 shows, for each of the first five policies, the average best cost divided by the average best cost obtained with the last combined policy per generation. As a first observation, the combination of all the genetic operations always leads in average to QEPs with associated costs several times lower than the other approaches. By nature, CS and S cannot solve critical structural shortcomings such as inefficient join order or inappropriate utilization of join implementations. Therefore, their application without being combined with other genetic operations results in high-costed QEPs which associated costs are, in general, several orders of magnitude higher than the costs obtained applying other genetic operations. CJ usually decreases very fast during the first generations, typically discarding *Nested-Loop Join* operations not using indexes, but it quickly converges yielding QEPs with associated costs which usually are about one order of magnitude higher than those obtained when it is combined with other genetic operations. RS outperforms CJ for 20 relations by randomly introducing modifications to the plans in all the dimensions although, after the first generations, it presents a slow convergence compared to other approaches. However, the analysis in [7] show that, when the number of relations increases, the search space grows exponentially and the probability of randomly finding a better solution rapidly decreases and, therefore, RS looses efficiency compared to other approaches. The execution of crossover operations (C) without any mutation operation presents a fast convergence, although after the first generations the quality of the results is generally several times higher than the cost of the best QEP yielded by the combination of all the genetic operations.

V. Related Work

Y. Tao et al. [10] tackle the large star-schema query problem by heuristically splitting a user-specified complex query, internally, into a form that can better utilize the capability of the underlying query optimizer for each block. However, the work is focused on studying a special type of complex query possessing a star-schema structure with snowflakes.

Genetic algorithms are a randomized search technique [11], [12] modelling natural evolution over generations using crossover, mutation and selection operations. Applied to query optimization, the first genetic approaches consider a reduced set of plan properties in crossover and mutation operations [13], [14], i.e. the amount of information per plan is very limited as plans are transformed to chromosomes, represented as strings of integers. This lack of information usually leads to the generation of invalid plans that have to be repaired. In addition, they do not explain how to deal with cyclic query graphs. A new crossover operation is proposed in [3] with the objective of making genetic transformations more aware of the structure of a database management system. Stillger proposed a genetic programming based optimizer that directly uses execution plans as the members in the population, instead of using chromosomes. However, mutation operations may lead to invalid execution plans that need to be repaired. A first genetic optimizer prototype was created for PostgreSQL [15], but its search domain is reduced to left-deep trees and mutation operations are deprecated, thus bounding the search to only those properties appearing in the execution plans of the initial population. Besides, execution plans are represented as strings of integers, thereby a lot of important information is lost. The *Carquinyoli Genetic Optimizer* (CGO) [7] is presented as the first sound optimizer based on genetic programming. Later in [16], a statistical model is presented to show that it is possible to establish a set of rules to parameterize a genetic optimizer for star join queries, independently of the random effects of the initial population.

Also, several variants of *random walk* algorithms have been proposed in [14], [17], [18], [19], [20], [21]. Randomized search techniques like Iterative Improvement or Simulated

Annealing try to remedy the exponential explosion of dynamic programming techniques by iteratively exploring the search space and converging to a nearly optimal solution.

VI. Conclusions

We present a comprehensive analysis of the behavior of genetic operations in CGO. Our main conclusions are as follows:

- Each mutation usually performs transformations on specific characteristics of the QEPs. The combined use of all the mutations opens the genetic optimizer to the traversal of the whole search space.
- Results show that, after a few generations, mutation operations tend to do useless work generating plans that are discarded without being used afterwards.
- The percentage of operations of each type used has an effect on the quality of the best plan. Although crossover operations are in general more powerful, the use of mutation operations is necessary to improve the quality of the optimizer as a whole.
- Some characteristics of the QEPs in the search space are better than others to obtain execution plans close to optimal.
- As a last conclusion, the combination of some mutations exploring orthogonal dimensions does not necessarily lead to improvements which are the addition of their independent gains, although they may be better.

The ideal mutation should generate low-costed QEPs, thus allowing plans to survive longer increasing their probability to introduce new information into the population.

The analysis presented in this paper gives an idea of the potential improvements that could be undertaken in a genetic optimizer. Our understanding is that genetic optimizers offer a whole bunch of opportunities to build a powerful tool to optimize large join queries.

Acknowledgment

The authors would like to thank the IBM Toronto Lab Center for Advanced Studies for supporting this research. The authors from UPC also want to thank Generalitat de Catalunya for its support through grant GRE-00352.

References

[1] P. G. 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 the 1979 ACM SIGMOD international conference on Management of data*. ACM Press, 1979, pp. 23–34.

[2] A. N. Swami and B. R. Iyer, "A polynomial time algorithm for optimizing join queries," in *Proceedings of the Ninth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 345–354.

[3] M. Stillger and M. Spiliopoulou, "Genetic programming in database query optimization," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 July 1996, pp. 388–393. [Online]. Available: http://www.dbis.informatik.huberlin.de/pub/papers/conferences/GP96.ps

[4] V. Muntés-Mulero, J. Aguilar-Saborit, C. Zuzarte, and J.-L. Larriba-Pey, "Cgo: a sound genetic optimizer for cyclic query graphs," in *Proc. of ICCS 2006*. Reading, UK: Springer-Verlag, May 2006, pp. 156–163.

[5] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang, "Query optimization in the ibm db2 family," *IEEE Data Eng. Bull.*, vol. 16, no. 4, pp. 4–18, 1993.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] V. Muntés-Mulero, J. Aguilar-Saborit, C. Zuzarte, V. Markl, and J.-L. Larriba-Pey, "Genetic evolution in query optimization: a complete analysis of a genetic optimizer," Dept. d'Arqu. de Comp. Universitat Politecnica de Catalunya (http://www.dama.upc.edu), Tech. Rep. UPC-DAC-RR-2005-21, 2005. [Online]. Available: http://www.dama.upc.edu

[8] 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 *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1984, pp. 1–8.

[9] D. J. K.A., "Introduction to the second special issue on genetic algorithms," *Machine Learning*, vol. 5, no. 4, pp. 351–353, 1990.

[10] Y. Tao, Q. Zhu, C. Zuzarte, and W. Lau, "Optimizing large star-schema queries with snowflakes via heuristic-based query rewriting," in *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2003, pp. 279–293.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Morgan Kaufmann, 1989.

[12] J. Holland, *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.

[13] K. Bennett, M. C. Ferris, and Y. E. Ioannidis, "A genetic algorithm for database query optimization," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, R. Belew and L. Booker, Eds. San Mateo, CA: Morgan Kaufman, 1991, pp. 400–407. [Online]. Available: citeseer.nj.nec.com/bennett91genetic.html

[14] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *VLDB Journal: Very Large Data Bases*, vol. 6, no. 3, pp. 191–208, 1997. [Online]. Available: citeseer.nj.nec.com/article/steinbrunn97heuristic.html

[15] PostgreSQL, "http://www.postgresql.org/." [Online]. Available: http://www.postgresql.org/

[16] V. Muntés-Mulero, J. A.-S. M. Pérez-Cassany, C. Zuzarte, and J.-L. Larriba-Pey, "Parameterizing a genetic optimizer," in *Proc. of DEXA '06*, September 2006, pp. 707–717.

[17] Y. E. Ioannidis and E. Wong, "Query optimization by simulated annealing," in *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1987, pp. 9–22.

[18] A. Swami and A. Gupta, "Optimization of large join queries," in *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1988, pp. 8–17.

[19] Y. E. Ioannidis and Y. Kang, "Randomized algorithms for optimizing large join queries," in *SIGMOD '90: Proc. of the 1990 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1990, pp. 312–321.

[20] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït, "On the effectiveness of optimization search strategies for parallel execution spaces," in *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 493–504.

[21] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten, "Fast, randomized join-order selection - why use transformations?" in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 85–95.