# CGO: A Sound Genetic Optimizer for Cyclic Query Graphs*

Victor Muntés-Mulero[1], Josep Aguilar-Saborit[1], Calisto Zuzarte[2], and Josep-L. Larriba-Pey[1]

[1] DAMA-UPC, Computer Architecture Dept., Universitat Politècnica de Catalunya, Campus Nord UPC, C/Jordi Girona Módul D6 Despatx 117 08034 Barcelona, Spain
{vmuntes, jaguilar, larri}@ac.upc.edu,
http://research.ac.upc.edu/DAC/DAMA-UPC
[2] IBM Canada Ltd, IBM Toronto Lab., 8200 Warden Ave., Markham, Ontario L6G1C7, Canada
calisto@ca.ibm.com

**Abstract.** The increasing number of applications requiring the use of large join queries reinforces the search for good methods to determine the best execution plan. This is especially true, when the large number of joins occurring in a query prevent traditional optimizers from using dynamic programming.

In this paper we present the Carquinyoli Genetic Optimizer (CGO). CGO is a sound optimizer based on genetic programming that uses a subset of the cost-model of IBM®DB2®Universal Database™(DB2 UDB) for selection in order to produce new generations of query plans. Our study shows that CGO is very competitive either as a standalone optimizer or as a fast post-optimizer. In addition, CGO takes into account the inherent characteristics of query plans like their cyclic nature.

## 1 Introduction

Query optimizers, which typically employ dynamic programming techniques [6], have difficulties handling large join queries because of the exponential explosion of the search space. Randomized search techniques remedy this problem by iteratively exploring the search space and converging to a nearly optimal solution. Genetic algorithms [2] are a randomized search technique that models natural evolution over generations using crossover, mutation and selection operations.

In this paper we present a genetic optimizer called the Carquinyoli[1] Genetic Optimizer (CGO) that is coupled with the DB2 UDB optimizer's cost model. CGO is an improvement over the work in [1] and [7] and proposes genetic operations that allow for cyclic plan graphs using genetic programming algorithms.

* Research supported by the IBM Toronto Lab Center for Advanced Studies and UPC Barcelona. The authors from UPC want to thank Generalitat de Catalunya for its support through grant GRE-00352.

[1] Name of a traditional Catalan cookie known for being very hard, which reminds us of the complexity of the large join optimization problem.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part I, LNCS 3991, pp. 156–163, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Our results show that the plans obtained by CGO equal those obtained by DB2 UDB for small queries, although the execution time of CGO is larger than that of DB2 UDB. When we turn to large queries, where DB2 UDB has to resort to heuristic approaches, CGO generates cheaper query plans, but the execution time of the optimizer is still larger. Note that DB2 UDB uses a greedy algorithm in those cases, since there is not enough memory to perform an exhaustive search using dynamic programming. In order to improve the optimization time, we propose a combined strategy that helps to improve the final plan obtained by DB2 UDB injecting the greedy plan in the initial population of CGO.

This paper is organized as follows. Section 2 introduces genetic optimization and describes CGO. Section 3 validates the genetic optimizer comparing it to DB2 UDB and proposes CGO as a post-optimizer for small databases. In sections 4 and 5, we explain the related work and draw some conclusions.

## 2 The Carquinyoli Genetic Optimizer

Inspired by the principles of genetic variation and natural selection, genetic programming performs operations on the members of a given population, imitating the natural evolution through several generations. Each member in the population represents a path to achieve a specific objective and has an associated cost. Starting with an initial population containing a known number of members, usually created from scratch, three operations are used to simulate evolution: *crossover operations*, which combine properties of the existing members in the population, *mutation operations*, which introduce new properties and *selection*, which discards the worst fitted members using a fitness function.

After applying these genetic operations, the population has been refreshed with new members. The new population is also called new *generation*. This process is repeated iteratively until a *stop condition* stops the execution. Once the stop condition 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 optimum execution plan in a vast search space. Each execution plan 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 execution plan. Intuitively, as the population evolves, the average plan cost of the members decreases.

**CGO.** CGO is a sound genetic query optimizer based on the ideas of genetic programming outlined above. CGO assumes that there is a parser that transforms the SQL statements into a graph. This graph contains a vertex for each referenced relation and edges joining a pair of vertices when a join condition between attributes of these relations appear in the query statement.

CGO also 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 the relational operations of PROJECT, JOIN, etc., and the root node

returns the final results of the query. The physical implementations of the operations used in the QEP are called plan operations and are described in more detail later in this section.

**Cyclic Query Graphs.** If we assume that a relation is never read twice, given $N$ relations in a query we exactly need $N-1$ join operations to join all the data during the query process. In a scenario without Cartesian products, this implies that, at least, we have $N-1$ explicit join conditions in the SQL statement, joining all the relations accessed in the query process. Of course, the number of join conditions can be larger, forcing some join operations in the tree to have more than one join predicate. A number of join conditions larger than $N-1$ implies cycles in the query graph.

In the presence of cyclic query graphs, two or more join predicates have to be used in some of the joins in the QEP. However, the same two conditions are not always merged together. Whether a join condition is located in a specific join operation depends on the order used to access the base relations.

One of the major enhancements of CGO is its capability to deal with cyclic query graphs. Our optimizer solves this problem merging the join predicates during the optimization process. Every join operation is associated with a single join predicate and is considered as an independent operation, giving versatility to our optimizer to handle cyclic query graphs.

**Merging Join Conditions.** Let us call $J_{i,j}$ a join operation with a single join predicate $Ri.a = Rj.b$. Every join operation has two input subtrees in the QEP, $S_i$ and $S_j$, from where tuples are retrieved to perform the join process. We say that a relation $R$ belongs to a subtree $S$, $R \in S$, if there is a scan operation in $S$ having $R$ as input relation.

An operation $J_{i,j}$ is merged with another existing single predicate join operation $J_{x,y}$ with subtrees $S_x$ and $S_y$ if

$$(R_i \in S_x \wedge R_j \in S_y) \vee (R_i \in S_y \wedge R_j \in S_x) \tag{1}$$

In fact, by *merge* we mean that both conditions $J_{i,j}$ and $J_{x,y}$ become a single operation with two join predicates.

Figure 1 shows an example where join operation $J5 = J_{1,3}$ has to be inserted in an existing partial plan. In step 1 we first check whether the join relations of $J5$ appear in the existing subtree. Since both join relations are scanned in the same subtree, we have to look for the node $J_{x,y}$ with subtrees $S_x$ and $S_y$ satisfying condition 1 (step 2). When node $J1 = J_{3,2}$ is checked, condition 1 is not satisfied. However, the optimizer detects that both input relations in $J5$ are in $S_3$ of $J1$. Therefore, $J2 = J_{1,5}$, the root node in $S_3$ of $J1$, is checked next. As $R_1 \in S_1 \wedge R_3 \in S_5$ is satisfied, $J2$ and $J5$ are merged in step 3. After the insertion $J2$, can be considered to have two join conditions or predicates, $R1 = R3$ and $R1 = R5$. However, as shown before, these two join predicates can be separated in other execution plan configurations. With this simple procedure, we allow CGO to handle cyclic query graphs, which previous genetic optimizers were not able to do.

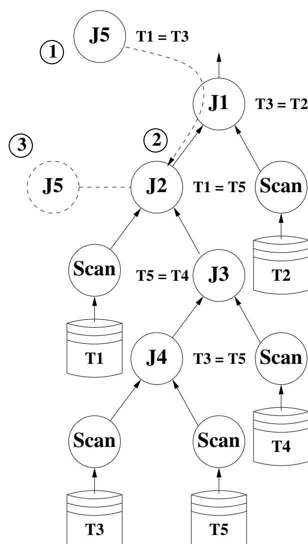


**Fig. 1.** Merging to join operations in an execution plan for a cyclic query graph

**Plan Operations.** The first aim of CGO is to outperform the most popular commercial optimizers when dealing with large join queries. Although commercial optimizers use a great variety of advanced techniques to improve the performance of DBMSs, for the sake of simplicity, CGO only considers a reduced set of these advanced techniques, such as prefetch or bit filters. The implementations considered by CGO are the *sequential scan* and the *index scan* for scan operations, the *Nested Loop Join*, *Merge-Scan Join* and *Hash Join* for join operations and two more unary operations, *Sort* and *Temp*, the first one for sorting data and the second one to materialize intermediate results in the QEP. Also, we reduced the language syntax to simplify our optimizer. Therefore, CGO can read simple SQL statements including multi-attribute selections, attribute constraints, equijoin conditions and sorting operations.

**Cost Model.** The cost model used to evaluate QEPs in CGO is an adaptation of the cost model in the DB2 UDB optimizer, in order to be comparable with existing optimizers. Overall, we use a fairly complete and comparable cost model based on CPU and I/O consumption.

**Genetic Operations.** In CGO, crossover operations randomly choose two QEPs in the population and produce two new trees preserving two subtrees from the parent plans and inserting the remaining operations one by one. Since each single-predicate join operation is handled separately, using this crossover method, the multiple join predicates in the QEP are automatically redistributed to the correct nodes and thus, we preserve the semantics of cyclic query graphs. A more detailed example can be found in [4].

Mutation operations are necessary to provide an opportunity to add new characteristics that are not represented in any of the execution plans in the

population. In this paper we assume that the whole search space is accessible if CGO grants the exploration of all the dimensions of this search space: tree morphology, join order in the QEP, join methods to be used in the join operations and scan methods to be used in the scan operations. We propose four different kinds of mutation operations for CGO; (i) Swap (S): a join operation is randomly selected and its input relations are swapped. This mutation grants potential access to all tree morphologies; (ii) Change Scan (CS): CGO randomly chooses a scan operation and changes the scan method if indexes are available, making possible the access to all the scan methods; (iii) Change Join (CJ): the optimizer randomly chooses a join operation and it changes its implementation to one of the other available implementations; (iv) 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, created a new and complete execution plan. This mutation grants the potential exploration of all the join operation orders.

S, CS and CJ do not take into account the occurrence of multiple join predicate operations since their transformations are not affected by the number of predicates. RS follows a construction method similar to crossover operations and thus, it allows for cyclic query graphs.

# 3 CGO Validation

In this section, we show that CGO is able to outperform a classical optimizer when this falls into heuristic algorithms because of a lack of memory, with large join queries. With this aim, we compare CGO with the optimizer of a well known commercial DBMS: the DB2 UDB optimizer. In order to verify the quality of the plans yielded by CGO we first compare the results obtained by CGO for queries based on the TPC-H Benchmark with query execution plans yielded by the DB2 UDB optimizer, that is, small join queries. Our results show, as expected, insignificant differences between the two optimizers as plotted in Figure 2 (left), although the optimization time of CGO is significantly larger than that of DB2 UDB. For more details, we refer the reader to [4]. Then, we compare CGO results for random-generated queries involving 20 and 30 relations, with those obtained by the commercial optimizer when it runs out of memory and uses its metaoptimizer, based on a sophisticated greedy join enumeration method. We always force the QEPs generated by CGO into the DB2 UDB optimizer, to avoid false results that are due to a possible bad integration of the DB2 UDB cost model. Finally, advanced features in DB2 UDB such as multidimensional clustering indexes, index ANDing and index ORing have been disabled.

## 3.1 Execution Details

We run two different test sets. In the first test set we generate 2 random databases and run 25 queries on each, where each query accesses 20 relations. In the second set, we create 4 random databases, running 25 random queries on

each, where each query accesses 30 relations. Therefore, in total we execute 150 random queries in 6 randomly generated databases. For both, we run CGO on populations containing 250 members, performing 150 crossover operations and 160 mutation operations per generation, through 300 generations. We present all the results using the scaled cost, meaning the cost of the QEP generated by the DB2 UDB optimizer divided by the cost of the QEP generated by CGO. The tool used to generate the random databases and queries is detailed in [4].

In the first test, CGO obtained significantly cheaper plans. CGO obtained plans that were on average 3.83 times cheaper than those obtained by the DB2 UDB optimizer. Again, note that some optimizations of DB2 UDB were turned off and the greedy join algorithm was used by this optimizer. There were only 2 cases where the DB2 UDB plans were cheaper than those obtained by CGO. In the second test, which involved 30 relational joins, ignoring the outliers, the average scaled cost was 4.09 times cheaper with CGO. In the very best cases, CGO reaches improvements of three orders of magnitude.

### 3.2 Combining CGO and DB2 UDB: A Fast Post-optimizer

The experiments show the potential of CGO to find low-costed QEPs in front of an heuristic search-based optimizer. However, the optimization time required by CGO is larger than the time required by an heuristic search. The amount of computation time, saved during run time, can clearly compensate for the increasing optimization time since, usually, for large join queries the total execution time raises to several hours and even days. We are currently working to reduce the optimization time. Preliminary results based on very simple memory optimizations show improvements over the 95 % in optimization time reduction.

For this reason, queries that require long processing times will obtain a large benefit by using CGO alone; however, small and fast queries cannot afford long optimization times. Therefore, we propose to use CGO in a post-optimization phase. We obtain the QEP yielded by the DB2 UDB optimizer and force it as one of the QEP in the initial population of CGO.

Figure 2 (right) shows the total cost evolution for the best plan in the population across time. For 20 relations, we obtain, on average, QEP which are 1.71 times better than the initial plan generated by the DB2 UDB optimizer after 30 seconds of execution. The slight cost regressions for higher times and the values below 1 are a consequence of the complex integration of the DB2 UDB optimizer cost model into CGO, which makes it very hard to calculate the exact same cost. Therefore, although CGO always finds better plans, they are costed differently in DB2 UDB. However, it is important to notice that we can never get a worse execution plan than the one generated by the DB2 UDB optimizer since we can always preserve it in case CGO does not find anything better in a limited amount of time. Trends for 30 relations are similar.

We would like to remark that if CGO were using the exact same cost model as DB2 UDB, the genetic optimizer would benefit from it, thus amplifying the differences between both optimizers, since CGO currently wastes time trying to optimize some QEPs which cost is overestimated later by the cost model of DB2 UDB.

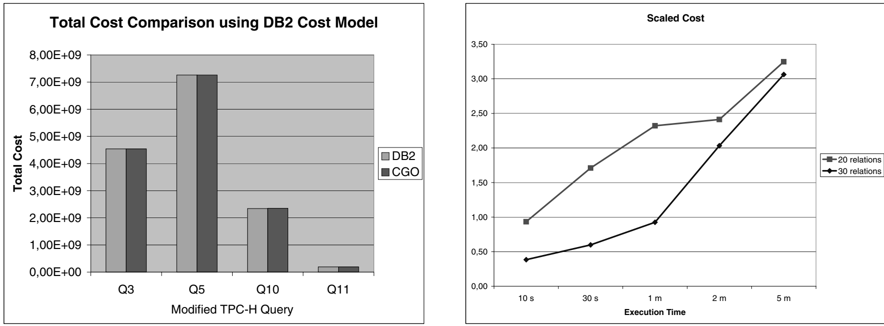


**Fig. 2.** (Left) Total Cost comparison for modified TPC-H queries using the cost model of the DB2 UDB optimizer. (Right) Evolution of the scaled cost of the best solution obtained by CGO across time.

## 4 Related Work

State-of-the-art query optimizers, which typically employ dynamic programming techniques [6], have difficulties handling large join queries because of the exponential explosion of the search space. In these situations, optimizers usually fall back to greedy algorithms. However, greedy algorithms, as well as other types of heuristic algorithms [9], do not consider the entire search space and thus may overlook the optimal plan, resulting in bad query performance, which may cause queries to run for hours instead of seconds. Randomized search techniques like Iterative Improvement or Simulated Annealing [3, 10, 7] remedy the exponential explosion of dynamic programming techniques by iteratively exploring the search space and converging to a nearly optimal solution.

Previous genetic approaches [1, 7] consider a limited amount of information per plan since these 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. A new crossover operation is proposed in [8] 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. Any of these previous approaches consider cyclic query graphs. A first genetic optimizer prototype was created for PostgreSQL [5], 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.

## 5 Conclusions

We presented CGO, a powerful genetic optimizer that is able to handle cyclic query graphs considering a rich variety of DBMS operations, reaching further

than all previously proposed genetic optimizers. For the first time in the literature, we implemented a genetic optimizer that allows us to compare its results with the execution plans of a commercial DBMS, DB2 UDB.

The main conclusions of this paper are: (1) CGO is a sound optimizer able to compete with commercial optimizers for very large join queries. (2) CGO, used as a post-optimizer, significantly improves the execution plans obtained by DB2 UDB in a short amount of time. And (3) CGO is capable of finding the optimal or a near-optimal execution plan in a reasonable number of iterations for small queries.

**Acknowledgment. IBM, DB2, and DB2 Universal Database** are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

## References

1. Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 400–407, San Mateo, CA, 1991. Morgan Kaufman.
2. J. Holland. Adaption in natural and artificial systems. The University of Michigan Press, Ann Arbor, 1975.
3. Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 9–22, New York, NY, USA, 1987. ACM Press.
4. Victor Muntes, Josep Aguilar, Calisto Zuzarte, Volker Markl, and Josep Lluis Larriba. Genetic evolution in query optimization: a complete analysis of a genetic optimizer. Technical Report UPC-DAC-RR-2005-21, Dept. d'Arqu. de Computadors. Universitat Politecnica de Catalunya (http://www.dama.upc.edu), 2005.
5. PostgreSQL. http://www.postgresql.org/.
6. P. Griffiths 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*, pages 23–34. ACM Press, 1979.
7. Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal: Very Large Data Bases*, 6(3):191–208, 1997.
8. Michael Stillger and Myra Spiliopoulou. Genetic programming in database query optimization. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 388–393, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
9. A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 367–376. ACM Press, 1989.
10. Arun Swami and Anoop Gupta. Optimization of large join queries. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 8–17, New York, NY, USA, 1988. ACM Press.