NON-EXHAUSTIVE JOIN ORDERING SEARCH ALGORITHMS FOR LJQO*

Tarcizio Alexandre Bini, Adriano Lange, Marcos Sfair Sunye,
Fabiano Silva and Eduardo Cunha de Almeida
Informatics Department, Federal University of Paraná, Centro Politécnico, Jardim das Américas, Curitiba PR, Brazil

Keywords: Query optimization, Join ordering, Randomized algorithms, Genetic algorithms.

Abstract: In relational database systems the optimization of select-project-join queries is a combinatorial problem. The use of exhaustive search methods is prohibitive because of the exponential increase of the search space. Randomized searches are used to find near optimal plans in polynomial time. In this paper, we investigate the large join query optimization (LJQO) problem by extending randomized algorithms and implementing a 2PO algorithm as a query optimizer in a popular open-source DBMS. We compare our solution with an implementation of a genetic algorithm. Through a multidimensional test schema, we discuss pros and cons about the behavior of these algorithms. Our results show that 2PO algorithm is fast to run and the costs of generated plans are better in most cases when compared to those of the genetic algorithms.

1 INTRODUCTION

Over the last 40 years, database management systems (DBMS) have experienced an enormous workload shift from transaction processing at kilobyte-scale to real-time data analysis at petabyte-scale. In the context of data warehouse, columns data storage such as in C-Store DBMS (Stonebraker et al., 2005), and map-reduce implementations like Hadoop/Hive (Thusoo et al., 2009), are becoming increasingly common both in commercial and academic area.

Storing data in columns in traditional DBMS is an onerous task, which can not bring major benefits (Abadi et al., 2008). In this storage architecture, each attribute is stored in a separated relation (Bruno, 2009). Thus, even a simple query that involves a few attributes demand several relational joins to compose its result. Costs expended in performing a large number of joins cause a performance degradation of the DBMS. In this context, the classic problem of join-ordering optimization is highly relevant.

Computationally finding the optimal join order that compose the execution plan is NP-hard (Ibaraki and Kameda, 1984). Dynamic programming techniques (Selinger et al., 1979), stay restricted to a small number of relations, approximately ten. Above this limit, it is considered a *large join query optimization* (LJQO) problem. Thereby, an optimization technique must consider a trade off between the generation of an optimal plan and the query execution.

Randomized algorithms can, in average, get good execution plans in polynomial time (Swami and Gupta, 1988; Ioannidis and Kang, 1991; Bennett et al., 1991; Louis and Zhang, 1998; Dong and Liang, 2007). However, an undesirable characteristic is the considerable costs variation (e.g., instability) of generated plans for the same query (Bini et al., 2009). Such instability is unacceptable in production applications, especially when the response time of a request is controlled. Furthermore, the insertion of a degree of uncertainty in the estimated time for a requested service or result can compromise the estimated time of a whole chain of processes depending on it. Thus, when we apply randomized algorithms to query optimization, the stability of generated plans is a decisive factor that must be considered.

In this paper, we address the LJQO problem through our implementation of the 2PO (Two-Phase Optimization) randomized algorithm. To evaluate the quality of generated plans and the execution time of the optimizers, we used a popular open-source DBMS (Postgresql, 2010). We created a wide select-projet-join queries range considering the joins graph complexity, number of joins and cardinality of the involved relations. In this context, we compare our

*Work partially funded by the Datluge CNPq-INRIA project.

Alexandre Bini T., Lange A., Sfair Sunye M., Silva F. and Cunha de Almeida E..
NON-EXHAUSTIVE JOIN ORDERING SEARCH ALGORITHMS FOR LJQO.
DOI: 10.5220/0003426501510156
In *Proceedings of the 13th International Conference on Enterprise Information Systems* (ICEIS-2011), pages 151-156
ISBN: 978-989-8425-53-9
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

solution with an implementation of the genetic algorithms. The results demonstrate the feasibility of applying our optimizer in commercial DBMS. Considering the execution time of the optimizers, 2PO outperformed genetic algorithm in all scenarios. 2PO optimizer also showed better quality of generated plans (low computational cost) in the most of analyzed queries.

This paper is organized as follows. Section 2 describes how the possible plans of a query can be organized considering important concepts of query optimization. Moreover, we present some randomized algorithms implemented and analyzed in this study. Section 3 describes the test methodology applied for evaluating the query optimizers. Section 4 presents the results obtained in our experiments. Finally, Section 5 concludes and presents future work.

2 BACKGROUND AND RELATED WORK

Query optimizer is the DBMS component that transforms a query written in a declarative language, like SQL, into a procedural *query execution plan* (QEP). There may be several QEPs corresponding to a same specified query. They are identical in result, but distinct in computational cost, like CPU time sharing, primary memory usage and disc access (Ibaraki and Kameda, 1984; Swami and Gupta, 1988). The set of all QEP that exhibit the same result is called *solution space* or *search space*. It is defined as the set of all possible *binary join trees* (BJT). The query optimizer task is to determine the QEP with lowest cost based on a cost function or model.

A common way to express graphically which relations mentioned in the query have join predicates is using an undirected graph called *join graph*. This structure can determine the complexity of finding the optimal join order. Nodes represent relations in a query. Edges represent join predicates between their respective relations. Regarding the possible forms of a join graph, five types are found in the literature: *chain*, *star*, *cycle*, *grid* and *clique* (Steinbrunn et al., 1997; Vance and Maier, 1996; Shapiro et al., 2001; Neumann, 2009).

Next, we describe the randomized and genetic algorithms, that can be an alternative to exhaustive searches when applied to LJQO problem.

2.1 Randomized Algorithms

The input of a randomized algorithm is another type of graph that represents the complete search space of a problem. Each node is a *solution* or *state* which is associated a cost defined by a specific cost function. Each edge is a possible *move* defined by transformation rules which allow that a state can be transformed into another.

Several apporaches were presented (Steinbrunn et al., 1997; Swami and Gupta, 1988; Ioannidis and Kang, 1990) to address the LJQO problem. Thus, in the Iterative Improvement (II) algorithm, QEPs can be represented as nodes in an undirected graph. Such nodes are inter-connected by edges which represent the transformations between these plans. The objective is to perform several movements between the graph nodes searching better solutions (execution plans) relative to its cost. II consists of several local optimizations, started randomly from different points of the search space, which are called *initial states* (Ioannidis and Kang, 1991). From each initial state, the algorithm traverses randomly the search space, always accepting moves if their costs are lower than the actual. This local optimization process is repeated until no further improvement can be found or a stopping condition is satisfied.

Another randomized algorithm applied to the query optimization problem is the Simulated Annealing (SA) (Ioannidis and Wong, 1987; Swami and Gupta, 1988). This solution was derived by analogy from the process of annealing of solids. Inspired by this physical process some terminologies, e.g., *temperature*, *freezing condition*, are used to orient the optimization process. Unlike the II, which uses several random initial states, the SA starts from a single state. During the optimization process, SA performs "*random walks*" always accepts neighbors states with lower costs. However, unlike the II, states with higher costs can also be accepted to a certain probability.

Ioannidis and Kang (Ioannidis and Kang, 1990) presented the 2PO (*Two-Phase Optimization*) algorithm to address the LJQO problem by combining the II and SA algorithms. In the first phase, the II is executed for a small period of time performing some local optimizations. This cover most of the search space and quickly reaches a state with low cost (called *local minimum*). The best local minimum found by II algorithm is used as a starting point for the SA, in the second phase. Then, the solution space is searched again for a state of even lower cost. Plans with lower costs than local minimum introduced by the II can be reached quickly through the SA.

2.2 Genetic Algorithms

Genetic Algorithms are search methods based on genetic and natural selection process. An important cha-

racteristic of this class of algorithm is that they do not work with a single solution, but with a set of solutions that is called *population*. These solutions are represented by *chromosomes*. Each chromosome is composed by genes that represent each part of the solution.

In joins optimization, the chromosomes correspond to the executions plans. Costs associated with it, corresponds to your adaptation degree to the environment. As in the search space of II and SA, the environment, corresponds to the set of possible solutions.

A genetic algorithm starts generating the first population of individuals randomly, with a fixed number of chromosomes. Chromosomes are selected (*selection*) from the population to become *parents*, based on fitness (Owais et al., 2005)[2]. The reproduction process occurs between pairs of selected chromosomes,through of the recombination between themselves (*crossover*) which produces the *offspring*. Some fraction of the population can be randomly chosen to have mutated a gene or a small set of them (*mutation*). The new population becomes the new generation and the process repeats itself (Bennett et al., 1991). Iterations are performed until improvements in the quality of the population are observed, or the demanded number of generations is reached or when the demanded solution is found.

In order to develop and analyze our implementation of 2PO optimizer, we used a popular open-source DBMS (Postgresql, 2010). This DBMS makes use of a genetic algorithm approach to enumerate possible BJTs. This algorithm called GEQO (*Genetic Query Optimization*) was presented by Martin S. Utesch (Postgresql, 2010) in 1997, as an alternative to the LJQO problem. More details about GEQO can be obtained in (Bini et al., 2009).

3 TEST METHODOLOGY

In this section, we detail our methodology to evaluate our implementation of the 2PO optimizer. First, we describe how the database was generated. After, how we developed the queries set for our experiments.

3.1 Database

The database schema and the query set are based on the systematic and multidimensional model proposed by Vance and Maier (Vance and Maier, 1996) et al. and Shapiro (Shapiro et al., 2001)]. One of its advantages is the independence of several construction parameters like number of relations and their cardinalities.

However, some parameters originally used by these authors were extended or reconfigured, in order to observe the behavior of the algorithms applied to LJQO problem. These reconfigurations were based on methodologies proposed by Swami and Gupta (Swami and Gupta, 1988), Ioannidis and Kang (Ioannidis and Kang, 1990) and Steinbrunn et al. (Steinbrunn et al., 1997).

The database is populed synthetically in accordance with the requirements of our experiments. Relations are created with their cardinalities ranging from 2^5 to 2^{19} as $log_2(2^{19}/2^5) = 14$ and mean $= 2^{12}$ (Vance and Maier, 1996).

All relations follow the same basic building layout, composed by three numerical attributes: an attribute *primary key* (*pk*) and others two attributes *foreign key* (*fk1, fk2*).

The tuples are inserted in the relations with values based on their cardinality. For the *pk* attribute, values are assigned sequentially from 1 to the relation cardinality. For the others attributes, *fk1* and *fk2*, the values are inserted at random, also following values from 1 to the relation cardinality, using an uniform distribution. Updates are never performed during the queries submission.

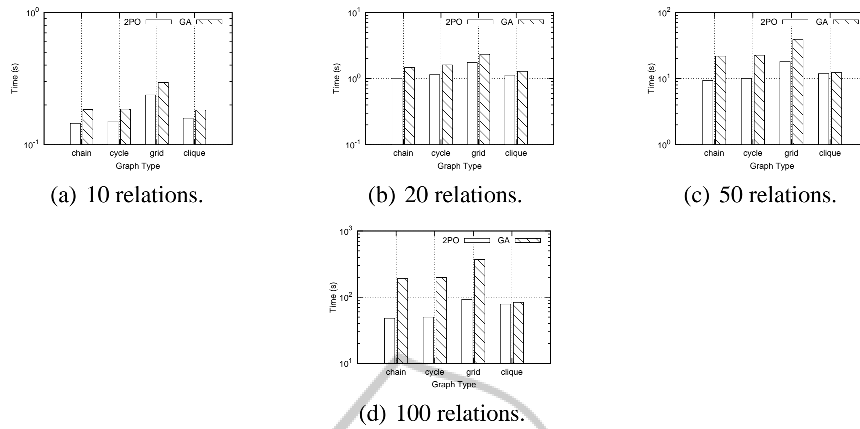
3.2 Generated Queries

Two steps are required to generate the query set: the *selection* relation sets and the *combination* of these relations in form of SQL queries. These steps are used for organization reasons and code reuse.

Selection. In the first step, 12 relation sets are selected deterministically according to the combination of two parameters: *Relations Number* and *LOGRATIO* (Vance and Maier, 1996; Shapiro et al., 2001). The Relations Number *N*, involved in the generated queries is respectively 10, 20, 50, and 100. LOGRATIO μ, is given by the logarithmic difference between the relation with greatest cardinality R_N and the relation with smallest cardinality R_1, namely, $log_2(|R_n|/|R_1|)$, represented by the values 6, 12 and 14. In each generated relation set, all selected relations are different, although their cardinality way be the same. For all relation sets, the geometric average of cardinalities was fixed at 2^{12}.

Combination. For each of the 12 selected sets, 40 queries are randomly generated, in order, 10 of them

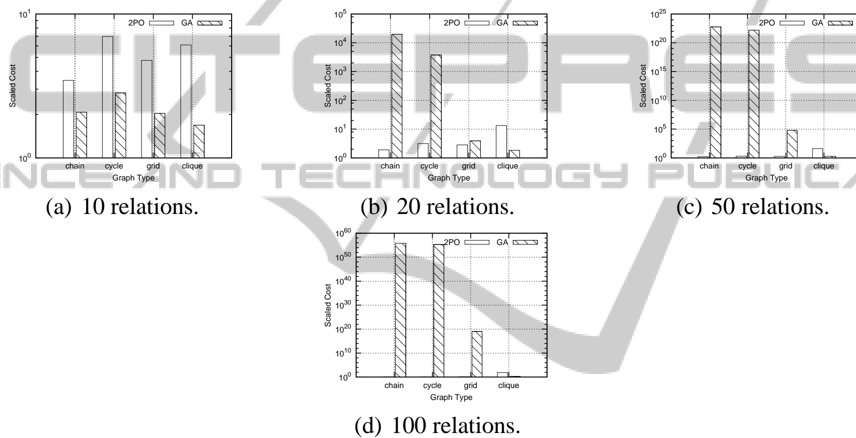
[2]Fitness - metrics to measure scheduler performance for each chromosome and that calculates the values for each one.



(a) 10 relations. (b) 20 relations. (c) 50 relations.

(d) 100 relations.

Figure 1: Optimization average time presented by 2PO and GA considering relations number and join graphs.



(a) 10 relations. (b) 20 relations. (c) 50 relations.

(d) 100 relations.

Figure 2: Average and scaled cost variation presented by 2PO and GA considering the relations number.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

for each join graph type chosen (*chain, cycle, grid* and *clique*) (Steinbrunn et al., 1997). Thus, the total number of generated queries is 480. Queries are generated as follows: from a relation set (R_1 to R_N), selected in the first step, each derived query follows a *random arrangement* of these relations (T_1 to T_N). From this arrangement, relations are "accommodated" in their SQL structures corresponding to each join graph.

4 EXPERIMENTS

This Section presents the comparison between our implementation of the 2PO optimizers and an implementation of the genetic algorithm. Results are presented using the scaled cost of plans, which is the ratio between its cost and the cost of the best plan found for the same query, independent from the optimizer. Since our test methodology is derived from a comparison among optimizers, the total execution time of the obtained plans was not computed. The plans obtained with the genetic algorithm, in some cases, presented extremely high costs. Thus, their executions were impossible, due to computational resources and time.

4.1 General Setup

Our experiments were conducted in a computer equipped with an Intel Xeon Quad-Core - 2GHz/64bits processor, with 12MB of L2 cache and 2GB/667MHz of RAM memory. As secondary memory, we used two SATA disks with 250 GB each, operating in RAID 0. The operating system employed was GNU/Linux kernel 2.6.24 X86-64.

The DBMS applied in experiments was the PostgreSQL 8.3. We used the plugin technique to compile code parts separately and then incorporate them into the DBMS as a library. Thus, we create the *LJQO plugin*, which includes our implementation of the 2PO algorithm. This plugin was equipped with a component to count the algorithms execution time. No changes in other DBMS components were required.

There are several configurations and control parameters that must be considered in the 2PO implementation. These parameters are further detailed in (Ioannidis and Kang, 1990).

We recall that PostgreSQL implements a genetic algorithm called GEQO. The GEQO module, in its default configuration, is designed to generate a nearly constant amount of plans for queries over seven relations. The 2PO in turn, tends to increase the optimization effort, as the number of relations increases. To eliminate this disadvantage, GEQO module was reimplemented to generate an equal plans amount to the 2PO optimizer. This configuration was performed individually for each query, considering the number of plans generated by 2PO in each case. In performed experiments, our implementation of GEQO module is called GA (*Genetic Algorithm*).

4.2 Performance Evaluation

The performance evaluation aims at verify the average time spent to optimize the queries. For this experiment, we consider 300 optimizations. This total is composed by 10 optimizations, for each one of 30 queries used, regardless of their LOGRATIO values.

Figure 1 presents the average time for optimization grouped by the relations number of each query. In queries with 10 relations (Figure 1(a)), the difference between the optimizers 2PO and GA was not significant. Although the number of generated plans by GA was approximately the same in relation to 2PO, its total time of optimization was relatively higher for queries chain, grid and cycle. In queries with 100 relations (Figure 1(d)), and join graph like chain, cycle and grid, the GA was almost 2 times worse than 2PO. On the other hand, in clique queries, its optimization time was very close from that presented by 2PO.

4.3 Costs of Generated Plans

Our final analysis considers the costs of generated plans by GA and 2PO optimizers. Figure 2 presents the average and scaled cost variation obtained by the optimizers considering the relations amount and the join graph of each query. In all graphs presented in this figure, scaled costs are arranged in logarithmic scale of base 10.

It is observed that both algorithms alternate in the generation of the best average of scaled cost. In chain, cycle and grid queries with relations greater or equal to 20, 2PO showed plans with superior quality (lower costs) in relation to presented by GA, as we can see in Figure 2 (b), (c) and (d). In this case, it is also observed that GA showed an accentuated degradation in the quality of the plans according increased the number of relations. In two particular cases, chain and cycle queries with 100 relations, respectively in Figure 2 (d), the quality of obtained plans by 2PO and GA was significantly large.

In Figure 2(b) to Figure 2(d), we can verify that the plans degradation obtained by the GA is strongly correlated with the number of edges of each join graph. It is noticed, the fewer is the number of edges, the worse is the quality obtained plans by the GA. Concerning the chain ($N-1$ edges) and cycle (N edges) queries, we can observe a slight improvement in the quality. In grid queries ($2N-3$ edges), the improvement is more evident. Finally, clique queries ($N(N-1)/2$ edges) with the maximum edges quantity, and obviously the join graph of greater connectivity, presented the best scaled cost, being better than those presented by 2PO.

Plans quality presented by 2PO was superior in almost all cases. However, two exceptions were identified. The first case refers to queries with 10 relations, independent of join graphs. The second, was for all clique queries, regardless the relations number.

5 CONCLUSIONS

Join optimization is part of query processing that represents a significant impact on relational DBMS efficiency. This paper presents this problem, with emphasis in LJQO.

We presented an implementation of a 2PO algorithm and compared with an implementation of genetic algorithm. Our solution proves to be more robust for most of the cases, mainly in queries with a large number of relations and low connectivity of their joins graphs. In this context, 2PO presented better plans compared to GA, and still, had a feasible optimization average time.

Our results were relevant in the actual context, since there is a growing demand for DBMS able to answer complex queries. Actually, such queries are really common in tools to generate management reports (OLAP - *Online Analytical Processing*) or deductive tools for Data Mining. Another potential source of complex queries are columns oriented DBMS (Stonebraker et al., 2005). In such applications, each attribute stored in a traditional DBMS is converted into an individual relation. Thus, queries read only the attributes that are required. Thereby, it is evident, the large number of joins required to compose the query results that involving several attributes in the SELECT clause.

There are still issues that need to be evaluated in

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

detail, how the use of aggregates, sort method (*ORDER BY*), and recursive queries. Even so, it is expected that these results can serve as a basis for the algorithms improvement and for developing new optimizations approaches.

Finally, 2PO algorithm is available as a plugin called LJQO. This plugin can be obtained via Internet[3] by interested in making improvements or analysis in our solution.

REFERENCES

Abadi, D. J., Madden, S. R., and Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 967–980, New York, NY, USA. ACM.

Bennett, K., Ferris, M. C., and Ioannidis, Y. E. (1991). A genetic algorithm for database query optimization. In *In Proceedings of the fourth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann Publishers.

Bini, T. A., Lange, A., Sunye, M. S., and Silva, F. (2009). Stableness in large join query optimization. In *ISCIS*, pages 639–644.

Bruno, N. (2009). Teaching an old elephant new tricks. In *CIDR*, pages 1–6.

Dong, H. and Liang, Y. (2007). Genetic algorithms for large join query optimization. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1211–1218, New York, NY, USA. ACM.

Ibaraki, T. and Kameda, T. (1984). On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502.

Ioannidis, Y. E. and Kang, Y. (1990). Randomized algorithms for optimizing large join queries. *SIGMOD Rec.*, 19(2):312–321.

Ioannidis, Y. E. and Kang, Y. C. (1991). Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. *SIGMOD Rec.*, 20(2):168–177.

Ioannidis, Y. E. and Wong, E. (1987). Query optimization by simulated annealing. In *SIGMOD Conference*, pages 9–22.

Louis, S. J. and Zhang, Y. (1998). An empirical comparison of randomized algorithms for large join query optimization. In *FLAIRS Conference*, pages 95–100.

Neumann, T. (2009). Query simplification: graceful degradation for join-order optimization. In *SIGMOD Conference*, pages 403–414.

Owais, S. S. J., Krömer, P., and Snásel, V. (2005). Query optimization by genetic algorithms. In *DATESO*, pages 125–137.

[3]http://git.c3sl.ufpr.br/gitweb?p=lbd/ljqo.git;a=summary

Postgresql (2010). Postgresql object-relational database management system. Available at URL: http://www.postgresql.org.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979). Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA. ACM.

Shapiro, L. D., Maier, D., Benninghoff, P., Billings, K., Fan, Y., Hatwal, K., Wang, Q., Zhang, Y., min Wu, H., and Vance, B. (2001). Exploiting upper and lower bounds in top-down query optimization. In *IDEAS*, pages 20–33.

Steinbrunn, M., Moerkotte, G., and Kemper, A. (1997). Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. (2005). C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 553–564. VLDB Endowment.

Swami, A. and Gupta, A. (1988). 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. ACM.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2:1626–1629.

Vance, B. and Maier, D. (1996). Rapid bushy join-order optimization with cartesian products. In *SIGMOD Conference*, pages 35–46.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS