Stableness In Large Join Query Optimization

Tarcizio Alexandre Bini, Adriano Lange, Marcos Sfair Sunye and Fabiano Silva
Universidade Federal do Paraná
Departamento de Informática, Curitiba - PR, Brazil
Email: {tarcizioab,adriano,sunye,fabiano}@c3sl.ufpr.br

***Abstract*—In relational database model, the use of exhaustive search methods in the large join query optimization is prohibitive because of the exponential increase of search space. An alternative widely discussed is the use of randomized search techniques. Several previous researches have been showed that the use of randomized sampling in query optimization permits to find, in average, near optimal plans in polynomial time. However, due to their random components, the quality of yielded plans for the same query may vary a lot, making the response time of a submitted query unpredictable. On the other hand, the use of heuristic optimization may increase stability of response time. This characteristic is essential in environments where response time must be predicted. In this paper, we will compare a randomized algorithm and a heuristic algorithm applied to large join query optimization. We used an open source DBMS as experimental framework and we compared the quality and stability of these algorithms.**

I. Introduction

Over the years, the use of relational DBMSs has become a viable alternative for the storage and retrieval of information in several areas. One of the main advantages of these DBMSs was to allow the increase of productivity through non-procedural query languages such as SQL [1]. However, the ambiguous nature of this kind of language, with respect to its execution form, introduced a planning problem [2]–[5]. The query optimization process aims then to enumerate possible execution plans and choose the one that minimizes the response time as well as the resources used for its processing. Due to the exponential growth of the search space, the use of algorithms that exhaustively enumerate all possible plans stay restricted to a small number of relations, approximately ten. Above this limit, we consider a large join query optimization problem (LJQO problem).

The algorithms applied to LJQO problem are divided into two main classes, heuristics and randomics. Steinbrunn *et al.* [6] make a comparison of these two classes of algorithms. In their work, the authors discuss the quality of plans generated by various algorithms in each one of these classes. The authors showed clear advantages of randomized algorithms to choose good plans. In addition, others works can be mentioned describing the quality of such algorithms [4], [5], [7], [8]. In spite of the good results demonstrated by randomized algorithms, these works present as result, the average cost or the best costs obtained by these algorithms.

Although a query optimization algorithm can, in average, get good plans, this average is more relevant on a global context than on individual cases. The use of randomized components in the query optimization as a method of listing and selecting possible plans can certainly generate good samples among the possible solutions. However, a undesirable characteristic of this technique is the considerable variation of generated plans for the same query. In environments where the response time of a request is controlled, the maximum time for getting a response or its variation is more important than its average value. Examples of this type of environment are common in the financial system, where transactions at the terminals have a time limit to respond. 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. Besides the estimated response time for a specific service, other factors such as consumption of resources by a process and simultaneous access to the same resources should also be considered. Assuming that the plans generated for a query can vary, the amount of resources required for its processing tends consequently to vary.

Based on these requirements, it is possible to evaluate if a DBMS has conditions to be deployed in such environments. The PostgreSQL [9] is an open code relational DBMS that is receiving attention as a viable alternative for the storage and retrieval of data. In this paper, we evaluated the PostgreSQL's query optimization process, considering the stability of plans generated for queries with a large number of joins. Currently, the PostgreSQL implements, for this class of queries, a genetic algorithm called GEQO (*Genetic Query Optimization*). As a way to evaluate this algorithm, we used the *Toolkit Database Test 5* (DBT-5) [10], which is a variant of *Transaction Processing Performance Council* E (TPC-E) [11] adapted to PostgreSQL. We also compared the plans generated by GEQO with a heuristic algorithm, which is an improvement of the work done by Guttoski *et al.* [12].

This paper is organized into sections as follows. Section II describes how the possible plans of a query can be organized. Section III details the basic structure for queries optimization in PostgreSQL and how the joins sequence of a query is created. In Section IV we present the genetic algorithm and its implementation in PostgreSQL. Then, Section V describes the heuristic algorithm used in our experiments as compared to genetic algorithm. Section VI presents the results obtained in our experiments. Finally, Section VII provides the conclusions obtained in the experiments and mentions future works.

978-1-4244-5023-7/09/$25.00 ©2009 IEEE

ISCIS September 14-16, 2009
METU
Northern Cyprus Campus

II. Query Optimization

The goal of the query optimization mechanism is to transform the internal representation of a query written in a non-procedural language into a procedural query execution plan (QEP). There may be several QEPs corresponding to a same query, so that each one can represent a different arrangement in both its algebraic composition as well as in the algorithms implemented for each algebraic operation. There are several studies that describe this plans enumeration problem [2], [4], [13], where the reader can get more details on its terminology. We will consider as query optimization the join ordering process in a binary join tree (BJT). Other algebraic operations like projections and selections are implicitly distributed in the tree using heuristics.

In our experiments, we used only selection-projection-join queries, so operations such as ordering, aggregations, grouping and distinct will not be considered.

III. Join Ordering in PostgreSQL

The basic structure of the PostgreSQL's join ordering process is the *RelOptInfo*. This structure may represent a base relation (base *RelOptInfo*), a sub-plan (intermediate *RelOptInfo*) or a complete plan (complete *RelOptInfo*). The construction process of a BJT consists in successive join operations among the base *RelOptInfos* and the intermediate ones until the formation of a complete *RelOptInfo*. The join operation between an intermediate and a base *RelOptInfo* represents a *left-deep join*. The join operation between two intermediate *RelOptInfo* represents a *bushy join*.

Each *RelOptInfo* has a *path list* indicating the possible corresponding physical access plans. Each path is internally represented as a tree of execution plan methods. In a join operation between two *RelOptInfos*, the paths of both operands are combined between themselves to generate the paths that represent the resulting *RelOptInfo*. In this operation, the total estimated cost for the execution of each path is calculated taking as base the *cost model*. Using this model, each operation receives a numerical value indicating the estimated amount of resources necessary to perform it.

Currently, PostgreSQL has two algorithms applied to the join ordering problem. The first, named *standard algorithm*, is based on System R [14]. This algorithm uses dynamic programming technique to enumerate all possible BJTs of a query, choosing the one that yield the cheapest estimated execution plan. The second algorithm, called GEQO, was proposed by Martin S. Utesch [15] as an alternative to the LJQO problem. The GEQO uses the genetic algorithm approach in order to enumerate possible BJTs from the algebraic space. Section IV will describe the GEQO in more details. In Section V we will describe a heuristic algorithm used in our comparative tests with the GEQO.

IV. Genetic Algorithm Implementation

Genetic Algorithms are search methods based on genetic and natural selection process [16]. An important characteristic of this algorithm class is that they don't 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.

The genetic algorithm starts generating randomly the first population of individuals with a fixed number of chromosomes. According to *fitness*, which is the degree of adaptation of an individual to the environment, the chromosomes are selected (*selection*) from the population to become *parents*. Reproduction occurs between pairs of chromosomes, then, existing 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 [8]. These 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.

Since the version 6.1, the PostgreSQL presents the GEQO as an implementation of genetic algorithms applied to queries optimization. In this implementation, each chromosome represents a possible complete join plan, where each gene makes reference to a base relation. Firstly, the algorithm starts a list of random chromosomes that will be the initial population of the optimization process. Each chromosome is then, transformed into a complete *RelOptInfo*. The lowest cost of each *RelOptInfo* generated is used as *fitness* value that will be employed to evaluate the improvement of the population. After that, the GEQO starts a series of selections and recombination in this population, excluding the worst individuals, until the maximum number of generations is reached. Finally, the GEQO selects the *RelOptInfo* of the best chromosome as the join plan to be used on query execution process.

The selection process is made choosing two chromosomes for each generation. The first chromosome is the one with lowest *fitness* value. The second one is randomly selected in the population.

The recombination process consists on constructing a directional graph called *edge_table*, where each node is a gene and each edge points to a predecessor or a successor of this gene in the chromosome. In this graph, both selected chromosomes are combined and the duplicated edges (*shared edges*) are marked. After that, a descendent chromosome (*offspring*) is generated by constructing a *tour* in this graph. First, a node is selected randomly. After, an iterative process selects each next step in the graph, choosing preferentially shared edges. If there are no shared edges in a node, a random function selects another possible edge in that node as next step in the *tour*. The process ends when all nodes were been reached only once.

Currently, GEQO uses an exponential function, upper bounded by a constant function, which defines the population size and the number of generations used in the optimization process. This function is defined by the Equation 1,

$$f(n) = \begin{cases} 2^{n+1}, & \text{if } 2^{n+1} \leq 50 * \textit{effort} \\ 50 * \textit{effort}, & \text{if } 2^{n+1} > 50 * \textit{effort} \end{cases} \quad (1)$$

where *n* represents the number of relations and *effort* is a parameter which can be defined by database administrator. As we can see, this function is upper bounded[1] by a constant value which depends on *effort* value. *effort* may vary between 1 and 10 and its default value is 5. In our experiments, we used the default value of *effort* and *n* varying from 11 to 19. Because of this, we considered $f(n) = 250$ for all performed tests.

V. Heuristic Algorithm Implementation

In order to carry out our experiments, we looked for a heuristic algorithm that could be competitive with GEQO on the quality of plans generated.

The first heuristic algorithm proposed as an alternative to LJQO problem was the Minimum Selectivity. This algorithm assumed that a good global solution depended on intermediate results with low cardinality. This algorithm works only in the left-deep trees search space. Therefore, the algorithm constructs a BJT through a incremental process, chossing at each step, the base relation that could represent the smallest estimated cardinality.

Another heuristic algorithm, similar to the Minimum Selectivity is the Top-Down. This algorithm assumed as a purpose that the latest join is the most important one in a join tree. As well as Minimum Selectivity, the Top-Down also uses only the left-deep trees as search space. However, this algorithm utilizes recursion, firstly selecting the relations that will form the last joins of the BJT. At each recursive step, the algorithm selects among those relations not yet selected, the one that represents the lowest join cost.

Both Minimum Selectivity as Top-Down are extremely simple. However, the plans generated by them are very poor in several cases [6]. Another interesting strategy for the joins ordering is the use of minimum spanning tree. Thus, a query is represented as a graph, where nodes are the base relations and edges are the joins between pairs of base relations. Krishnamurthy, Boral and Zaniolo [17] proposed an algorithm, called KBZ, based on a particular property of acyclic queries [2], which allows to find the optimal nested join order in polynomial time. This algorithm starts generating a minimum spanning tree of the base relations involved in a query. Then, each node of this tree is selected as root, ordering the relations in a process called *linearize*. The result of each ordering process represents a left-deep tree. Finally, the algorithm selects the cheapest left-deep tree generated.

Another algorithm based on minimum spanning tree was proposed by Guttoski *et al.* [12]. This algorithm, that we named KQO (*Kruskal Query Optimization*), is based on the Kruskal's minimum spanning tree algorithm as building strategy of BJTs.

[1]There also is a lower bound in Equation 1 which is not applied in our experiments and because of simplifications, it will not be presented.

Firstly KQO builds an edge list representing the possible joins between each pair of base relations. For each edge, an intermediary *RelOptInfo* is also created representing the join between these relations. The weight of each edge is assigned to the lowest cost among the *paths* of its respective *RelOptInfo*. The edge list is ordered, and serves as basis for the BJT construction process. Successively, KQO selects the edge with lowest weight as the first subtree of the plan. After this, the algorithm continues selecting the subsequent edges, discarding those that represent cycle on the join graph. During this process, new subtrees may be created or new base relations are added to the existing subtrees, so that does not exist any base relation in common between the subtrees. If an edge represents the union of two subtrees, both must be united in a single subtree. For each subtree generated, a corresponding *RelOptInfo* is created. The KQO construction process ends when all edges are visited and a single tree representing the complete *RelOptInfo* has been created.

Differing from the other heuristic algorithms presented, KQO can build *bushy trees* as join plans. However, the characteristic of the Kruskal's algorithm applied to join ordering process, allows KQO to select preferably *left-deep trees*. Due to these features, we selected the KQO as part of our experiments, considering however some changes. As the KQO is a greedy algorithm, the order of the edge list is crucial for constructing a BJT. We modified KQO to avoid the generation of edges which might represent cartesian products. According to comparative tests already performed, we observed a large improvement in quality of plans regarding to the edge generator proposed by Guttoski *et al.* [12].

VI. Results

We carried out our experiments in a computer equipped with an Intel Xeon Quad-Core - 2GHz/64bits processor, with 12MB of L2 cache and 2GB/667MHz of RAM memory. The operating system used was GNU/Linux kernel 2.6.24 X86-64 and the PostgreSQL version used was 8.3.

In order to evaluate the quality of the execution plans generated by KQO and GEQO, we used TPC-E benchmark methodology, properly adapted to the needs of our experiments. Provided by *Transaction Processing Performance Council* (TPC) [18] this methodology presents a *Online Transaction Processing* (OLTP) workload, that is composed by multiple transactions simulating a brokerage firm environment [11].

The TPC-E database size is defined by CUSTOMER table cardinality. In our tests, we used the factor of 1000 records stored in this table. This is the minimum value generated by TPC-E methodology. Although small, the choose of this cardinality demonstrated be enough to carry our experiments.

The tool used to implement TPC-E methodology was the DBT-5 open-source toolkit, created by Rilson O. do Nascimento [19] in association with the *Open Source Development Labs* (OSDL) organization, now Linux Foundation. We used this tool to perform the data load of TPC-E into a PostgreSQL database.

TPC-E benchmark executes a transaction series (12 in the whole), which make inserts, updates and deletions of information data in the database. Due to these transactions perform data changing, compromising then the comparison between the generated plans by each algorithm, we decided to discard the operations that might perform any modification on involved tables.

The TPC-E originally does not provide queries with a significant number of relations. Therefore, we created a set of nine queries which permitted us to evaluate the behavior of the algorithms in LJQO problem. Basically, each query that was utilized in our tests, extracts several information from the benchmark's database, such as, data regarding the brokerage firm and tax rates applied to negotiations. The number of relations mentioned in each of the nine created queries is presented in the title of each graphic of Figure 1.

During the execution tests, the queries were randomly subjected 30 times to the database using KQO and GEQO algorithms. We applied such pratice to extract all presented results.

A. Generated Plans

Figure 1 presents the estimated plan cost in relation to a series of performed measurements. Each graph presents a query with a distinct number of involved relations. The vertical axis indicates the scaled cost of obtained plans in relation of optimal plan.

In Figure 1, we can see that GEQO presented, in queries Q1, Q2 and Q3, better plans than KQO for all measures. In query Q1, over 65% of plans generated by GEQO presented near-optimal costs. However, we can see in this query some variation of generated plan costs.

The queries from Q4 to Q8, the genetic algorithm generated execution plans with a large variation. In query Q7, GEQO presented plans 15 times worse in relation to optimal plan. This variation becomes more evident when the number of involves relations increases. We observed that the random characteristic of GEQO in creating and improving plans is insufficient to generate a regular sampling of search space. In addiction, in some cases the obtained plans might be many distant of optimal solution, becoming impracticable.

In query Q9, which involves 19 relations, the KQO algorithm presented lower cost compared with all plans obtained by GEQO. In this same query, we can also verify the greatest variation of plans generated by GEQO in all evaluated queries.

In the worst cases, the KQO presented costs 3.52 times higher in relation to the optimal solution. A relevant factor of KQO was its regularity in all generated plans for each optimized query.

The Figures 2 and 3 presents a summary of data obtained in the Figure 1. In these graphs, the queries Q1 to Q9 are arranged in "x" axis in ascending order by the relations number involved in each query.

Figure 2 presents the graph with scaled cost average of each submitted query. We can verify in this figure that KQO presented in average, execution plans with lower cost compared to

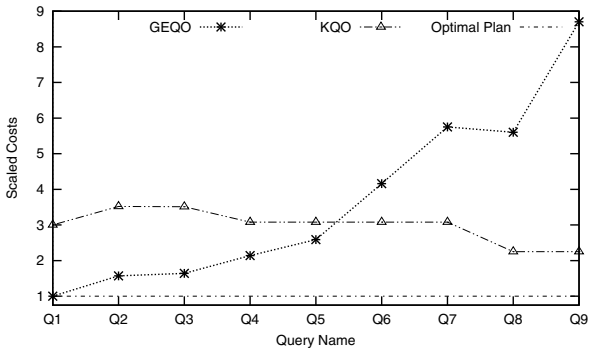

Fig. 2: Scaled Cost Average

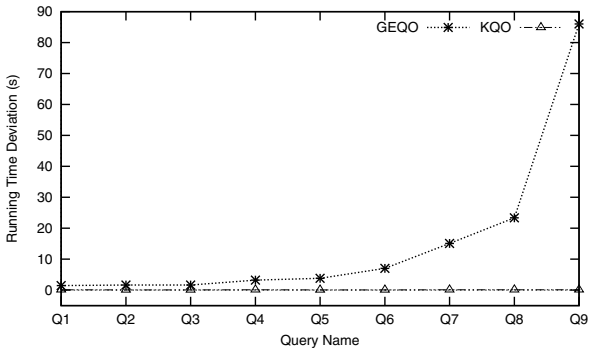

Fig. 3: Standard Deviation of Running Time

GEQO when the number of relations referenced by the query is bigger than 15. Over this number, we observed a significant increase in the average of execution plans costs presented by GEQO optimizer.

In the Figure 3 we have the graph that shows the standard deviation of the total running time presented by KQO and GEQO optimizers, considering the amount of mentioned relations in each query. In this graph we can verify that GEQO demonstrates trend to show variations in the queries execution time in relation to average cost. This feature becomes more evident from query Q5, which presents 15 relations.

B. Execution Time

The test results shown in Figure 4 demonstrate the average time in milliseconds needed to execute the optimizer algorithms.

We can verify in Figure 4, that in all evaluated queries, the KQO had its execution time significantly lower than GEQO. In general, both algorithms showed a linear evolution in their execution time with respect to the number of involved relations.

Although it demonstrated a constant behavior in the Figure 4, the KQO had a small variation in the execution time, from 1.1 ms in query Q1 to 2.1 ms in query Q9. On the other hand, GEQO showed an accented evolution of its execution time of 74.9 ms in query Q1 to 213.6 ms in query Q9.

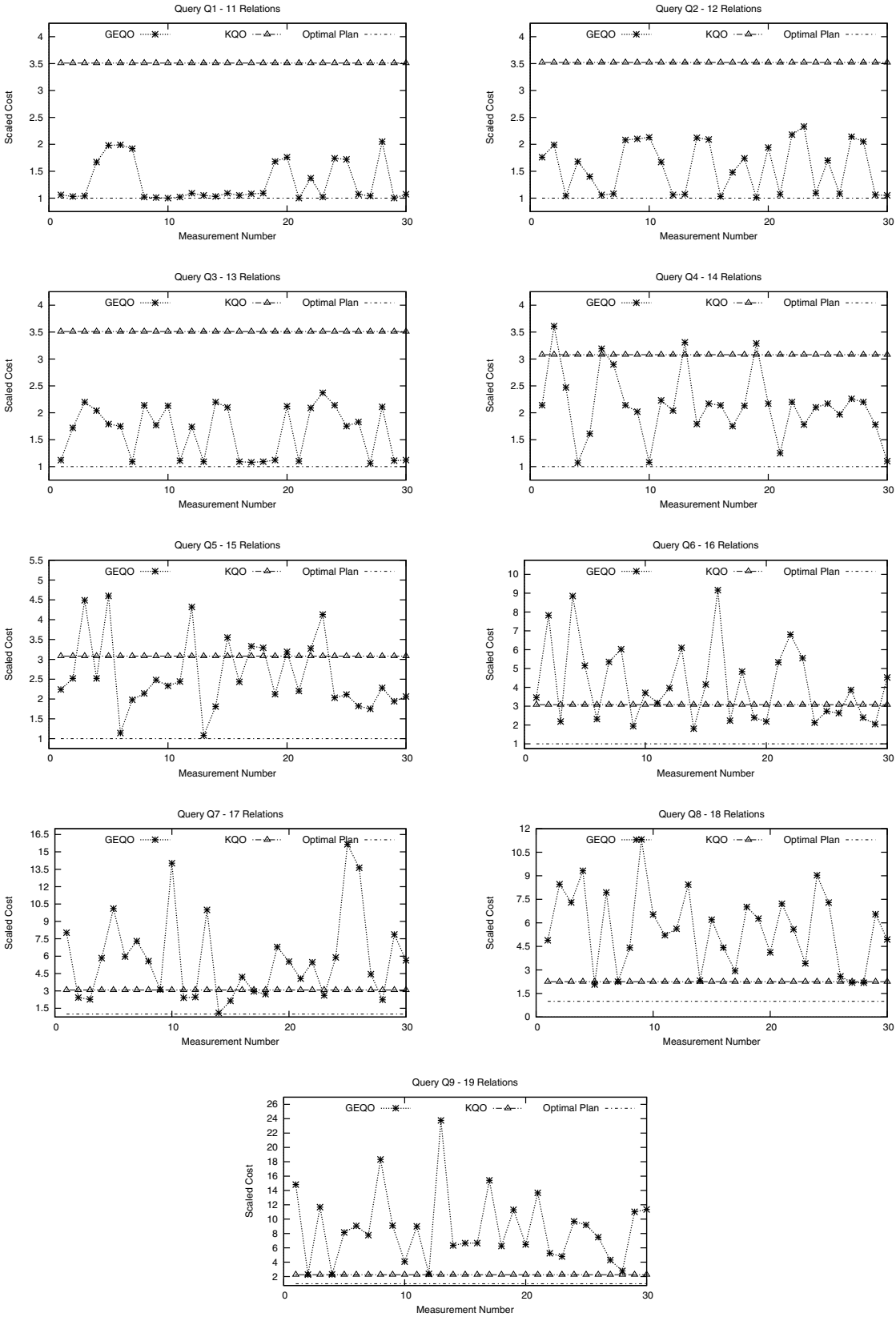

Fig. 1: Queries Scaled Cost vs Measurement Number

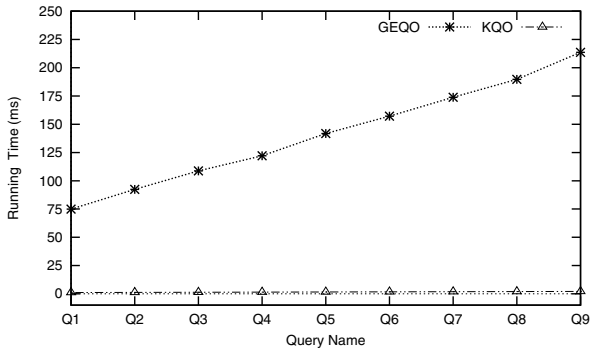

Fig. 4: Optimization Time Average

VII. Conclusions

In controlled computational environments, the response time of requested services must be fixed or obey known limits. Common examples of such environments are in financial system as banking operations or outsourced services. In these environments, inserting components that have a great variation of response time may compromise a whole chain of related services. Moreover, the propagation of this uncertainness is not restricted to affected direct services, but indirect reflexes like CPU time sharing, primary memory usage and disc access requests must be considered.

Based on these requirements, we verified the behaviour of PostgreSQL within controlled environments in processing large join queries. Considering this, we evaluated the GEQO behavior using repeated executions of a set of queries, and as result we verified a great variation of generated plans. In addition, we also verified that this variation increased progressively with the number of involved relations. Finally, we demonstrated the effects of this variation over the total running time of submitted queries.

As comparison with GEQO, we selected a heuristic query optimizer that was applicable to LJQO problem. KQO demonstrated a simple technique and a suitable heuristic for our experimental environment. In our experiments, we observed that KQO obtained plans around three time of optimal plan.

Based on these experiments, we demonstrated that the PostgreSQL's query optimization mechanism is currently unsuitable to process large join queries within controlled environments. In such environments, PostgreSQL is restricted to queries involving a few number relations, where the exhaustive search algorithm may be applicable. Over this limit, we consider more adequate heuristic planning techniques, even if in some cases, the quality of the generated plans may be lower than using randomized algorithms.

Although inadequate in these conditions, some improvements may be applied on GEQO in order to improve its predictability in generating plans. Dong and Yiwen [7] consider some heuristic techniques as a way to bias the generation of initial population as well as the selection mechanism of genetic algorithms applied to LJQO problem. The authors observed some improvements on yielded plans in certain query classes. Similar heuristics might be applied on GEQO in order to reduce the effect of randomized components over the generated plans. In addiction, further algorithms applied to LJQO problem may be evaluated to improve the predictability and stability of PostgreSQL within controlled environments.

References

[1] M. M. Astrahan and D. D. Chamberlin, "Implementation of a structured english query language," *Commun. ACM*, vol. 18, no. 10, pp. 580–588, 1975.

[2] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing n-relational joins," *ACM Trans. Database Syst.*, vol. 9, no. 3, pp. 482–502, 1984.

[3] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 111–152, 1984.

[4] A. Swami and A. Gupta, "Optimization of large join queries," *SIGMOD Rec.*, vol. 17, no. 3, pp. 8–17, 1988.

[5] Y. E. Ioannidis and Y. Kang, "Randomized algorithms for optimizing large join queries," vol. 19, no. 2. New York, NY, USA: ACM, 1990, pp. 312–321.

[6] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *The VLDB Journal*, vol. 6, no. 3, pp. 191–208, 1997.

[7] H. Dong and Y. Liang, "Genetic algorithms for large join query optimization," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2007, pp. 1211–1218.

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

[9] "PostgreSQL, object-relational database management system," available at URL: http://www.postgresql.org.

[10] "DBT5-Tollkit Database Test 5," available at URL: https://osdldbt.svn.sourceforge.net/svnroot/osdldbt/trunk/dbt5/.

[11] Transaction Processing Performance Council, "TPC BENCHMARK E - Version 1.6.0," Transaction Processing Performance Council, Tech. Rep., 2008.

[12] P. Guttoski, M. Sunye, and F. Silva, "Kruskal's algorithm for query tree optimization," in *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, Sept. 2007, pp. 296–302.

[13] Y. E. Ioannidis, "Query optimization," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 121–123, 1996.

[14] P. Selinger, "Access path selection in a relational data base system," *ACM-SIGMOD Conference on Management of Data*, 1979.

[15] The PostgreSQL Global Development Group, "PostgreSQL 8.3.1 Documentation," PostgreSQL Global Development Group, Tech. Rep., 2008.

[16] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Alabama: Addison-Wesley Publishing Company, 1989.

[17] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of nonrecursive queries," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 128–137.

[18] "TPC-E - OLTP," available at URL: http://www.tpc.org/tpce/ default.asp.

[19] R. O. D. Nascimento, M. Wong, and P. R. M. Maciel, "DBT-5: A fair usage open-source TPC-E implementation for performance evaluation of computer systems," p. 9, 2008.