An Evolutionary Multi-Agent System for Database Query Optimization

Frederico A. C. A. Gonçalves
Graduate Program in Electrical Engineering - Federal University of Minas Gerais
Belo Horizonte, Brazil
IT Center - Federal University of Ouro Preto
Ouro Preto, Brazil
fred@nti.ufop.br

Frederico G. Guimarães
Department of Electrical Engineering - Federal University of Minas Gerais
Belo Horizonte, Brazil
fredericoguimaraes@ufmg.br

Marcone J. F. Souza
Department of Computer Science - Federal University of Ouro Preto
Ouro Preto, Brazil
marcone@iceb.ufop.br

ABSTRACT

Join query optimization has a direct impact on the performance of a database system. This work presents an evolutionary multi-agent system applied to the join ordering problem related to database query planning. The proposed algorithm was implemented and embedded in the core of a database management system (DBMS). Parameters of the algorithm were calibrated by means of a factorial design and an analysis based on the variance. The algorithm was compared with the official query planner of the H2 DBMS, using a methodology based on benchmark tests. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with low execution costs in the majority of the cases.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems; H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software; I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search

Keywords

Join Ordering, Multi-agent System, Query Optimization

1. INTRODUCTION

Database Management Systems (DBMS) are computer systems that have an essential role in the creation, storage and maintenance of information. These systems are among the most complex software systems, integrating a number of components that must work together to guarantee the correct operation of the whole environment.

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

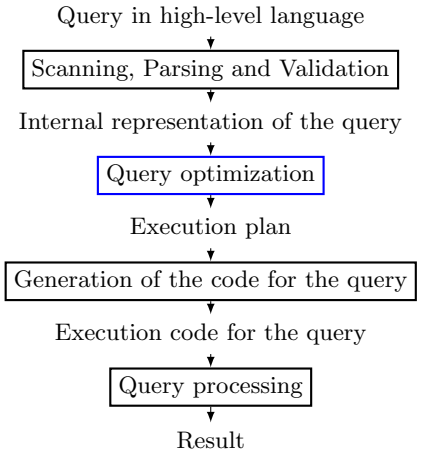

Figure 1: Typical steps for the processing of a query.

According to [9], the processing of one query can be divided into the steps shown in Figure 1. In this paper, we focus on the query optimization step. The task of this component is to define the execution plan for the query. The definition of the execution plan can involve an ordination of many types of database operations: UNION, INTERSECTION, DIFFERENCE, JOIN and so on. Specifically in the case of the join operations, the most time-consuming operation in query processing, the task can be viewed as a combinatorial optimization problem commonly known as join ordering problem. It consists in defining the best order of execution of the join operations among the relations specified in the query to be processed. The joining between two relations is an operation in relational algebra allowing the merging of information from these relations. The join operation will be treated as a binary operation (*two-way join*). However, this operation can have more than two relations as input too (*multiway-join*), and in accordance with [9], the combinations for multiway joins grow very rapidly.

Regarding query optimization, Ioannidis [16] proposes two classes of spaces: the algebric space and the space of structures and methods. The algebric space refers to the exe-

cution order of the actions considered, using relational algebra as representation (e.g. σ for data selection). The space of structures and methods, on the other hand, relates to the implementation choices available for the DBMS in hand. Possible choices include: sequential access method or index-based, nested-loop join methods, hash-join methods, or merge-join methods, among others. The different ways of representing the solutions combined with the access and join methods define different possibilities for join ordering, which implies in general an NP-complete problem [15]. A more detailed description of the join ordering problem can be found in [15, 25, 16, 23].

According to Garcia-Molina et al. [11], the I/O operations dominate the time spent in query processing. Given this observation, we can estimate the cost of a given query in terms of the number of accesses to disk and I/O operations. In addition to the accesses to disk, the CPU operations can also be considered in the cost model of the queries.

Query optimization and planning have a direct impact in the response time of the system, since a good execution plan would lead to a faster query result. In this article we present an algorithm for query optimization based on a multi-agent system. The main feature of the algorithm is to have a team of intelligent agents working together in a cooperative or competitive way to achieve the solution of the problem. Such systems differ from other parallel systems by presenting interaction between agents. These agents have characteristics such as reactivity, proactiveness, sociability, and so on [29]. An application of cooperation between agents for decision support can be found in [5].

More recently, some researchers have explored the integration between evolutionary algorithms and multi-agent systems, trying to take the best from both worlds. Evolutionary algorithms work with a population of structures that encode possible solutions for the problem [4, 7]. This population evolves iteratively by means of heuristic operators inspired or motivated by concepts of natural systems and Darwinian principles. In a typical evolutionary algorithm, the fitness of the individuals depends only on the quality of that individual in solving the problem. The integration between multi-agent systems and evolutionary algorithms lead to the so-called Evolutionary Multi-Agent Systems (EMAS). In such systems, the agents have also the ability to evolve, reproduce and adapt to the environment, competing for resources, communicating with other agents, and making autonomous decisions [8]. An example of EMAS is available in [2, 1], where the authors employ the use of Memetic Algorithm and Multi-Agent system.

There is some work in the literature regarding the join ordering problem, see Section 2. Many of these papers propose algorithms to solve the problem in an exhaustive or non-exhaustive way. Exhaustive approaches guarantee the optimal planning, while non-exhaustive methods do not. Some well-known DBMS such as PostgreSQL[1] and H2[2], employ exhaustive methods in query processing when the query involves a small number of relations. For queries that extrapolate a specified limit, due to the complexity of the problem, a non-exhaustive method is used.

The present article describes an approach that can be classified as a non-exhaustive one. As far as we know, EMAS have not been applied to such a context. Therefore, the main contribution of this article is the development of an evolutionary multi-agent system included directly in the query planner of the H2 DBMS. The agents of the system are able to interact and evolve in parallel. Additionally, the algorithm extends the behavior of purely evolutionary methods, with the inclusion of refinement and constructive heuristics in the set of actions of the agents. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with low execution costs in the majority of the cases.

The article is organized as follows. Section 2 presents and discusses the related work. The proposed algorithm is detailed in Section 3. Section 4 reports the calibration of the parameters of the algorithm and the results obtained with some benchmark tests. Section 5 presents our conclusions.

2. RELATED WORK

The join ordering problem has been studied in the literature for more than three decades. The seminal work is presented by [21], advancing an exhaustive method based on dynamic programming (DP) of a relational database system called System-R. The algorithm exhibited a time complexity $O(N!)$, and became a classic approach for solving the join ordering problem, where N stands for the number of relations in the query.

In [15], the authors demonstrate that this problem belongs in general to the class of NP-Complete. The authors used the Nested-Loop-Join as the join method. They presented two algorithms A and B, with time complexity $O(N^3)$ and $O(N^2 \log N)$, respectively. The algorithm B is applicable only to acyclic queries and guarantees the optimal solution, while the algorithm A is not restricted only to acyclic queries and does not guarantee the best solution.

An extension of Algorithm B [15] is presented by [19]. In this work, the authors developed an algorithm with time complexity $O(N^2)$ known as KBZ algorithm.

An algorithm based on the simulated annealing (SA) meta-heuristic is presented in [18] for query optimization involving many relations. The algorithm found some optimal solutions in some problems and got close to the best solutions in other test problems.

Many methods are compared in [25]: Perturbation Walk, Quasi-random Sampling, Iterated Improvement(II), Sequence Heuristic and Simulated Annealing (SA). The authors create a test database to evaluate the heuristics, considering cardinality, selectivity of the join predicates, distinct values and indices. Moreover, test queries were generated with directed graphs. According to the results, the heuristic II performed best. In a continuation of the work, another methods were tested in [24], they are: II, SA, KBZ, Greedy Algorithm and so on. According to the results, the II and the *Greedy Algorithm* performed better.

Based on the algorithms II and SA, an algorithm called *2PO* has been proposed in [17], it makes use of the two techniques mentioned. The *2PO* showed be more efficient than II and SA in the solutions quality and optimization time.

A genetic algorithm applied to the join order problem is presented in [4]. Two different strategies for crossover and mutation methods were used. The genetic algorithm was compared against the DP [21], and was capable to find better solutions to more complex problems.

[1]PostgreSQL: www.postgresql.org
[2]H2 Database: www.h2database.com

To solve some limitations of the KBZ, the authors in [26] proposed a extension of the algorithm (AB) with some improvements and a time complexity $O(N^4)$. The computational experiments compared: AB, Greedy Algorithm, DP, KBZ, II and $2PO$. The AB returned, on average, better solutions than the non-exaustive techniques, and was able to find solutions as good as those obtained by DP.

The meta-heuristic Tabu Search (TS) [13] is explored in [20]. Three methods were tested: TS, SA and II. The TS method found better solutions in almost all test problems, only for less complex problem the algorithm was worst than others.

Another exhaustive algorithm named *blitzsplit* was developed by [28]. The algorithm is based on DP and show a time complexity $O(3^N)$. According to the authors, the proposed algorithm found solutions in short time and little effort was expended to eliminate very costly solutions.

A comparative between many algorithms (DP, KBZ, SA, II, $2PO$, Genetic algorithms, among other methods) is presented in [23]. The results show that exhaustive methods fit better in less complex problems. The $2PO$ presented in general better results over solutions quality and optimization time, while SA was capable to find better solutions spending more optimization time. For low optimization times, the II, Genetic and 2PO were more indicated.

Genetic algorithms have been applied again in [7]. The authors conclude that the efficiency of the method is directly related to the cost model for the queries and also the neighborhood structures and operators used in the algorithm.

In [12], the authors present a multi-agent system for query optimization in distributed DBMS. The authors compare their method with dynamic programming, showing that their approach takes less time for processing the queries. They define the following agents: Query Distributor Agent (QDA), Local Optimizer Agents (LOAs) and Global Optimizer Agent (GOA). QDA divides the query into sub-queries and distributing them to the LOAs, that apply a local genetic algorithm. GOA is responsible for finding the best join order between the sites.

An extension of the algorithm proposed by [12] is given in [30, 10]. The extension focuses on building an adaptive system. This adaptive system presented a reduction of up to 29% in time response. A new agent called Adaptive Query Agent (AQA) was added to the system. This agent recognizes the similarity between queries. When similarities are identified, previously stored execution plans are re-used, otherwise, the query is executed and the agent's knowledge is updated. The adaptive agent also manages stored plans.

A parallel based DP algorithm is discussed in [14]. The algorithm was prototyped in the DBMS *Postgresql*, version 8.3. The algorithm improved the optimization time over classic DP [21] up to 547 times. Beyond that, for some problems, it obtained a linear speedup.

In relation to some DBMS in the market, the following strategies can be cited:

- **H2** - The version 1.3.167 from 2012[1], uses a brute force method for queries with up to 7 relations. For queries with more than 7 relations, a mixed algorithm composed by an exhaustive, greedy and random search methods is applied;
- ***Postgresql*** - The version 9.2 has an optimizer based on DP [21] for ordination of queries with up to 11 relations and a genetic algorithm for queries with more than 12 relations;
- ***Mysql***[2] - This DBMS has a hybrid algorithm based on greedy and exhaustive techniques. The exhaustive search is controlled and the greedy methodology is applied to evaluate promising solutions.

[1]www.h2database.com/html/performance.html

[2]MySQL: www.mysql.com

3. QUERY OPTIMIZER BASED ON EMAS

In this paper we developed an evolutionary multi-agent system for the join ordering problem. EMAS can extend the behavior of conventional genetic and multi-agent algorithms. The agents can act in proactive and reactive way, they can make decisions by applying differentiated interaction mechanisms with the other agents and explore the solutions space in a smart way by using the traditional operators of the genetic algorithms.

The environment is composed by a set of agents, where each agent works to find the best solution, i.e. the best join order for the relations in the query. The system can be described as $EMA = \langle E, \Gamma, \Omega \rangle$, where E is the environment, Γ is the set of resources, Ω contains the types of information available for the agents in the system. The environment is accessible, which means that the agents can obtain precise and complete information about the state of the environment. It is also non deterministic, for there is no certainty about the state resulting from a given action, and dynamic, since the agents can modify the environment during execution.

The environment E can be described as $E = \langle T^E, \Gamma^E, \Omega^E \rangle$, where T^E represents the topography, Γ^E are the resources and Ω^E is the information. We use only one kind of resource and information. The resource is expressed in terms of life points of each agent. The information about the other agents and the best current solution is represented in Ω^E.

The topography is given by $T^E = \langle A, l \rangle$, A represents the set of agents in the environment and l is a function returning the localization of a specific agent. Each agent a can be described by the tuple $a = \langle sol^a, Z^a, \Gamma^a, PR^a \rangle$, sol^a is the solution represented by the agent, actually encoded as an integer vector. Each element in the vector represents a relation in the query. The vector defines the join order for processing the query. Z^a is the set of actions of the agents. Γ^a and PR^a define the life points and profile of the agent. The set of system actions is detailed below:

- **getLife** - Used to obtain life points from another agent;
- **giveLife** - Give life points to another agent;
- **findWorse** - Search for an agent with worse fitness value;
- **findPartner** - Search for a partner in the set of agents in the environment;
- **crossover** - Crossover operator used to generate offspring;
- **mutation** - Mutation operator;
- **updateBestSolution** - Update the best solution;

- **randomDescent** - Random descent local search;
- **bestImprovement** - Best Improvement Method;
- **semiGreedyBuild** - Construct a solution by using a semi-greedy heuristic;
- **processRequests** - Evaluate and process pending requests;
- **changeProfile** - Change Profile of the agent;
- **die** - action of dying, when the life points of the agent ends.

Each agent is associated with an execution thread. These agents can be classified as hybrid ones, since they are reactive with internal state and deliberative agents seeking to update the best solution and not to die. The following characteristics can be highlighted: Reactivity, Pro-activity, Social ability and Veracity. The mutation and local search operators employ swap movements as illustrated in Figure 2.

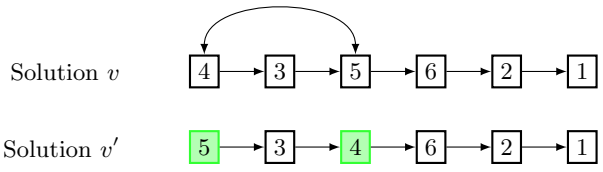

Figure 2: Swap movement - N^S

Two crossover operators are available: *Ordered Crossover* - OX [6] and *Sequential Constructive Crossover* - SCX [3]. When the crossover action is executed, one of these operators is selected at random. The examples described next use as input the solutions in Figure 3.

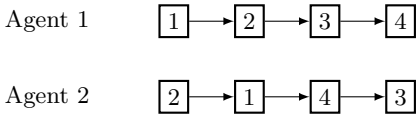

Figure 3: Initial agents

In the crossover OX, a crossover point is selected at random, which defines the part of Agent 1 that goes to the descendent. The rest of the sequence is taken from Agent 2 in an ordered way avoiding repetition of elements. Considering the agents in Figure 3 and a crossover point after the second relation, the resulting descendent is presented in Figure 4.

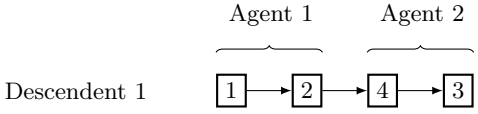

Figure 4: Descendent OX

In the strategy SCX, the cost of the join operations between each pair of relations is stored in a cost matrix. The values in the cost matrix are based on the cost model currently used in H2. The method starts by adding the first relation from Agent 1 to the descendent. After that, the descendent inherits the next relation from the parent with the smallest cost in the given cost matrix. Ties are solved randomly. For the agents in Figure 3, the descendent is given in Figure 5.

Cost Matrix				
	R1	R2	R3	R4
R1	99	10	20	5
R2	10	99	15	35
R3	20	15	99	45
R4	5	35	45	99

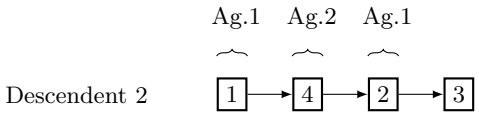

Figure 5: Descendent SCX

The action semiGreedyBuild is based on a semi-greedy (greedy-random) heuristic, which introduces randomness into a greedy constructive heuristic. A new solution is constructed by inserting relations step by step. For each insertion, the candidate elements are sorted according to a greedy function. In this work, we adopt two different greedy functions, based on two classification criteria (selected at random). The first one is based on the number of links between each relation to the other in the query. The relations with less links are better classified by the greedy function. The other approach is based on the cost of links. The relations with the smaller costs are better classified by the greedy function. The best elements form the restricted candidate list (RCL). The new element to be added to the solution is randomly taken from the RCL. When all relations have been inserted, a new solution is returned.

In a multi-agent system, the agents can execute different goals, which can be: survive, generate a solution together with another agent, refine his own solution and so on. To accomplish that, different profiles were designed to different goals, they are:

- **RESOURCE** - The agent tries to increase his life points by searching agents with worse fitness values and requesting part of their life points;
- **REPRODUCTION** - The agent looks for partners to produce new solutions by crossover;
- **MUTANT** - The agent suffers mutation;
- **RANDOM_DESCENT** - The agent refines its own solution by using random descent local search;
- **SEMI-GREEDY** - The agent builds new solutions by using the semi-greedy heuristic;

The Multi-Agent Query Optimizer (MAQO) pseudo-code is presented in the Algorithm 1. This algorithm first initializes the system (line 2). Then, three agents are created with the profiles RANDOM_DESCENT, SEMI-GREEDY

and MUTANT respectively (lines 4, 6 and 8). The initial solution used by the agent RANDOM_DESCENT is the same order given by the query, that is, the order in which the relations are read during the initial parsing of the query. The other solutions are constructed using the number of links as greedy function. The initialization of the agents and final refinement of the solution are done in lines 14, 16 and 18. The Best Improvement Method guarantees that the returned solution will be a local optimum in relation to the neighborhood N^S.

Algorithm 1: MAQO

```
   Data: QueryRelations
   Result: sol*
1  begin
2  |  Environment E ← initializeEnvironment();
3  |  //Create at least one agent with the profile
   |  RANDOM_DESCENT;
4  |  createAgent(E, RANDOM_DESCENT);
5  |  //Create at least one agent with the profile
   |  SEMI-GREEDY;
6  |  createAgent(E, SEMI-GREEDY);
7  |  //Create at least one agent with the profile
   |  MUTANT;
8  |  createAgent(E, MUTANT);
9  |  //Create the other agents with the profile
   |  REPRODUCTION;
10 |  for i=4 to E →MAXAGENTS do
11 |  |  createAgent(E, REPRODUCTION);
12 |  end
13 |  //Initialize threads;
14 |  initializeAgents(E);
15 |  //Wait threads to finish;
16 |  waitAgents(E);
17 |  //Best Improvement on Best Agent;
18 |  E →bestAgent→bestImprovement();
19 |  return E →bestSolution;
20 end
```

Once the agents have been initialized, the evolutionary process will also start, since agents may evolve through the execution of the related actions, which can be individual or joint actions. Each agent executes the goal of each profile by performing actions associated to that profile. We present next how the set of actions is distributed in each profile:

- **RESOURCE** - findWorse, processRequest, getLife, changeProfile and die;
- **REPRODUCTION** - findPartner, processRequest, crossover, updateBestSolution, changeProfile and die;
- **MUTANT** - mutation, processRequest, updateBestSolution, changeProfile and die;
- **RANDOM_DESCENT** - randomDescent, processRequest, updateBestSolution, changeProfile and die;
- **SEMI-GREEDY**- semiGreedyBuild,processRequest, updateBestSolution, changeProfile and die;

At every iteration of the agent in a given profile, its life points decrease. When an agent has no more life points, it executes the action die. When all agents have died, the algorithm stops. The action processRequests depends on the kind of request made. For instance, if an agent requests life points from another agent, the action processRequest will execute the action giveLife in response to the request. Changing profile is allowed only after a minimum number of iterations in the current profile or when the life level gets to a predetermined critical limit. In this case, the new profile becomes RESOURCE.

4. EXPERIMENTS

The computational experiments were performed in a *Core i7-2600 CPU 3.40*GHz, with 16 GB *RAM*, operational system *Ubuntu* 10.10-x86_64. The algorithm was developed in Java and IDE Netbeans. The DBMS H2 version 1.3.167 was used in the implementation. In the literature, deterministic evaluation methodologies are found in [28, 22] and random methodologies in [25, 24]. Benchmarks for evaluation of database systems were made available by the corporation Transaction Processing Performance Council[1] (TPC). The design of experiments proposed here is based on [25, 24]. The database was created and loaded according to the following information:

- **Cardinality distribution of tuples**- [10, 100) - 20%, [100, 1000) - 64%, [1000, 10000] - 16%;
- **Distribution of distinct values of the tuples** - [0, 0.2) - 75%, [0.2, 1) - 5%, 1 - 20%;
- **Relations Columns** - three per relation, being one reserved for primary key;
- **Percentage of indices and foreign keys** - 25% and 25%, respectively.

The number of relations presented in the queries was fixed in: 30, 50, 80 and 100 relations. Besides, the generation of the test queries was guided by four different graph shapes: chain, grid, star and multi-star. Actually, 150 relations were built using different values by selecting them randomly in the intervals described previously, but all test queries involve 100 relations maximum. The test queries were generated at random, varying the selection of relations and the selection of columns to form the join predicate. In all cases, a set of 10 queries in graph shape was created. It is worth noting that a series of sanity checks are executed to allow the creation of the foreign keys correctly. An example of a query in chain format is presented as follow:

Listing 1: Chain - SQL query

```
SELECT * FROM R1, R7, R2, R10, R150, R50 WHERE
    R1.col1 = R7.col2 AND R7.col3 = R2.col2 AND
    R2.col1 = R10.col3 AND R10.col1 = R150.col3 AND
    R150.col2 = R50.col1
```

The cost model adopted is the same defined by the DBMS H2. For this reason, we considered only the method Nested-Loop-Join. We remark that the DBMS H2 considers only the I/O operations in its cost model. Moreover, the possible representations for the solutions in the search space is restricted to left-depth tree. H2 also does not include in its model values related to intermediate results of the join operations, as presented in [27]. Therefore, the cost of a given join depends on the cardinality of the relations in the

[1]TPC: http://www.tpc.org

join operation, neglecting the intermediate results. More information about common restrictions adopted to limit the exploration of the solutions space are available in [17, 23, 11].

4.1 Calibration of the algorithm

In order to calibrate the parameters of the algorithm, we have selected 5 test queries with 30 relations in the form multi-star graph. We have used a factorial design of fixed effects. The factors analyzed are: the maximum number of agents and initial life percentage. The life of the agents is given by this percentage times the number of relations in the query. Table 1 describes the levels for each factor.

Table 1: Factors and levels

Factor	Level
Maximum number of agents	8, 16, 32, 64
Initial life	1, 2, 3, 4

We have considered two complete replications, totaling 160 observations. The number of observations is determined as follows: $N_t = N_c \times N_q \times N_r = 4^2 \times 5 \times 2 = 160$, where N_c is the number of possible combinations, N_q is the number of queries, and N_r is the number of replications. The response variables are the cost of the generated plan and the setup time. The confidence interval was set as 95%, implying a significance level $\alpha = 0.05$. All observations were randomized. The test hypotheses are:

1. $\mathbf{H_0}$: $\tau_1 = \tau_2 = \ldots = \tau_a = 0$ (no effects for the factor Maximum number of agents);
   $\mathbf{H_1}$: at least one $\tau_i \neq 0$.

2. $\mathbf{H_0}$: $\beta_1 = \beta_2 = \ldots = \beta_b = 0$ (no effects for the factor Initial life percentage);
   $\mathbf{H_1}$: at least one $\beta_j \neq 0$.

3. $\mathbf{H_0}$: $(\tau\beta)_{11} = (\tau\beta)_{12} = \ldots = (\tau\beta)_{ab} = 0$ (no interaction effect for the factors);
   $\mathbf{H_1}$: at least one $(\tau\beta)_{ij} \neq 0$.

The statistical analysis was done with (ANOVA - Analysis of Variance). According to the results obtained in this experiment, there was no significant factors with respect to the setup computational time and the cost of the generated plans. Regarding the cost, all generated plans achieved the same cost. The parameters chosen were the ones associated with the minimum setup time. A summary of the results generated by the statistical software *Minitab* with respect to the setup times is given in Table 2. In this table, DF represents the *Degrees of Freedom*, SS represents the *Sum of Squares*, MS means *Mean Square*, F and P refer to the type and p-value of the statistical test.

The selection of values for the parameters was based on Figure 6. Based on this experiment we have chosen 32 agents and life equal to 1. The generated model did not violate any of the established assumptions.

From the experiments, we note that some plans considered better with respect to the cost, actually took more time to process the query than other plans with higher costs. This behavior shows that the cost model of the DBMS H2 does not fit well with the real costs.

Table 2: ANOVA - Setup time of the generated plan

Source	DF	SS	MS	F_0	P
agents	3	22239	7413	0,62	0,613
life	3	21532	7177	0,60	0,625
agents×life	9	114160	12684	1,06	0,441
Error	16	191915	11995		
Total	31	349847			

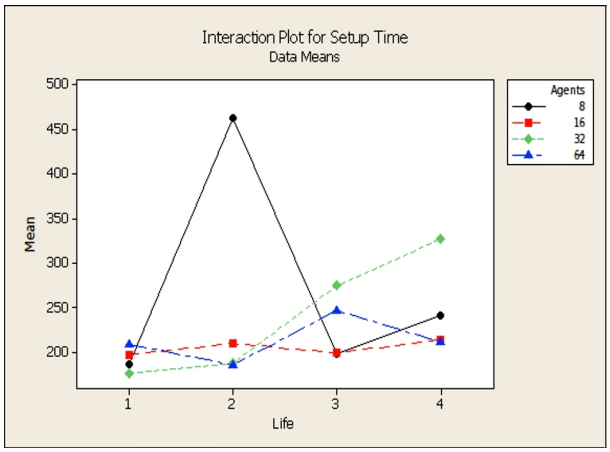

Figure 6: Setup time - Interaction of factors

4.2 Results

For the analysis of the results, we have used again a factorial design of fixed effects. Two different algorithms were tested, the default algorithm in H2 (H2 Query Optimizer - H2QO) and the developed evolutionary multi-agent algorithm (MAQO). Another factor studied was the format of the generated queries. Moreover, 3 complete replications were executed, giving a total of 480 executions to each planner. The output variable for data analysis was the cost of the generated plans. The confidence interval was set as 95%.

The total of executions of each algorithm is given by the following equation: $N_t = N_c \times N_q \times N_r = 16 \times 10 \times 3 = 480$, where N_c is the number of possible combinations, N_q is the number of grouped queries and N_r is the total of replications. The hypotheses are:

1. $\mathbf{H_0}$: no effect for the main factors and his interactions for all levels;
   $\mathbf{H_1}$: effect for the main factors and/or his interactions for at least one of the levels.

For the test queries, the cost increases as the complexity of the problem increases. For this reason we study the behavior of each algorithm in each dimension separately. The average costs were calculated in logarithmic scale. This was done to avoid that the results recorded in the DBMS get greater than the maximum limit in H2. The query timeout was set to 30 minutes. It is noteworthy that normality, independence and homoscedasticity assumptions were not violated by the generated models. The results were similar for 30, 50, 80 and 100 relations. Due to limited space, only the results

for 100 relations will be presented. All the results will be available from the authors [1].

4.3 Queries with 100 relations

Table 3 shows that the factors (and their interaction) have significance on data variability. The difference in the costs is caused by the algorithms and the shape of the queries. Consequently, the null hypothesis can be reject with 95% degree of confidence. Besides, the determination coefficient $R-Sq(adj)$ means that the generated model totally explains of the variability.

Table 3: ANOVA - 100 relations

Source	DF	SS	MS	F_0	P
Blocks	2	0,1	0,0	1,27	0,3
Alg.	1	3430,4	3430,4	96798,87	0,0
Shape	3	33301,8	11100,6	313239,46	0,0
Alg.Shap.	3	1697,3	565,8	15965,38	0,0
Error	14	0,5	0,0		
Total	23	38430,1			

S = 0,188250 R-Sq = 100,00% R-Sq(adj) = 100,00%

Based on Figure 7, we can verify that the average cost of the factor Algorithm in the level MAQO is lower than in the level H2QO. This results shows the superiority of the developed algorithm. Regarding the format of the queries, the GRID shape returned lower costs, while the MULTI-STAR shape was the most difficult to solve. It is also observed in Figure 8, that the evolutionary algorithm reached lower costs in all shapes.

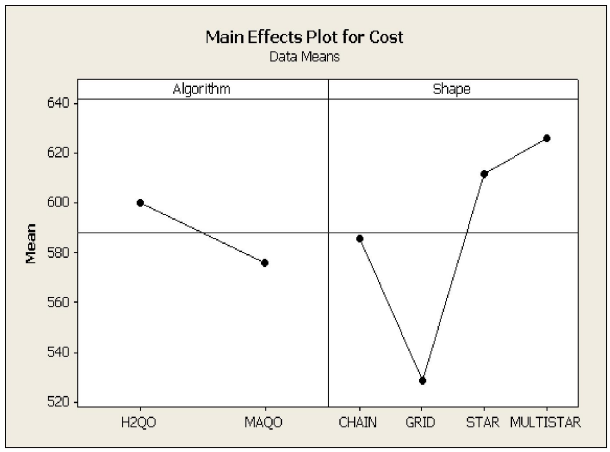

Figure 7: Main Effects - 100 relations

4.4 Discussion

It is possible to observe that the developed algorithm is able to obtain better results in terms of the execution cost of the plans in all cases. Additionally, out of 480 executed queries, only 1% of plans generated by the default planner in H2 were better than the ones generated by MAQO.

Table 4 presents an overview of the results obtained by both algorithms. It is clear that the multi-agent algorithm performed better in all test problems.

[1]http://cpdee.ufmg.br/∼zepfred/gecco2013/

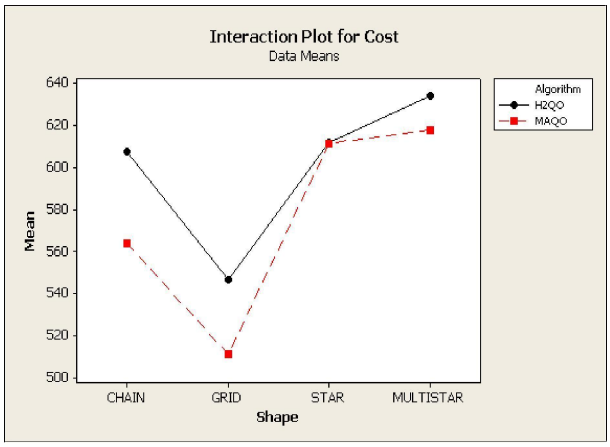

Figure 8: Interaction Effects - 100 relations

Table 4: Best Plans Overview

Query Shape	MAQO(%) - H2QO(%) 30	50	80	100
Chain	93 - 7	100 - 0	100 - 0	100 - 0
Grid	100 - 0	100 - 0	100 - 0	90 - 10
Star	100 - 0	100 - 0	100 - 0	100 - 0
Multi-star	97 - 3	100 - 0	100 - 0	100- 0

5. CONCLUSIONS

This article presented an algorithm based on evolutionary multi-agent systems. A number of actions and profiles for the agents were defined. We developed a benchmark of test queries following ideas from [25, 24] to evaluate and compare the developed algorithm and the default planner available in the DBMS H2. The test problems involve queries with 30, 50, 80 and 100 relations. The shapes of the representations adopted for the queries were graphs of the following types: chain, grid, star and multi-star.

The proposed algorithm was implemented in the optimization core of the DBMS H2. Another factorial design was used to compare MAQO and the official planner H2QO. Based on the results of the experiments, it is possible to say that MAQO was able to find low cost plans in 99% of the cases. The results suggest the superiority of the proposed approach and the benefit of using an evolutionary multi-agent system for query optimization and planning. However, it was observed that the cost model in H2 does not fit well with real costs of the plan. For this reason, the algorithm proposed has achieved longer execution times in some problems, even though the estimated cost was lower. Because of this fact, only the costs of the plans were considered.

We believe that the inclusion of the intermediate results in the join operations and the use of statistical information might improve the coherence of the cost model with the real cost of the queries.

Given the results reported, one can conclude that the developed algorithm is better than the official planner in H2. By using a more accurate cost model, the efficiency of the proposed query optimizer would be self-evident, representing real reduction in execution times.

6. ACKNOWLEDGMENTS

The authors would like to thank the support given by NTI UFOP, CAPES, CNPq (grants 305506/2010-2 and 306458/2-010-1), FAPEMIG (grants PPM-469-11, APQ-04611-10 and PPM-00366-12).

7. REFERENCES

[1] G. Acampora, J. Cadenas, V. Loia, and E. Ballester. Achieving memetic adaptability by means of agent-based machine learning. *Industrial Informatics, IEEE Transactions on*, 7(4):557–569, 2011.

[2] G. Acampora, J. Cadenas, V. Loia, and E. Ballester. A multi-agent memetic system for human-based knowledge selection. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 41(5):946–960, 2011.

[3] Z. H. Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics and Bioinformatics*, 3:96–105, 2010.

[4] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. In *International Conference on Genetic Algorithms*, pages 400–407, 1991.

[5] K. Dahal, K. Almejalli, and M. A. Hossain. Decision support for coordinated road traffic control actions. *Decision Support Systems*, 54(2):962 – 975, 2013.

[6] L. Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 136–140, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

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

[8] R. Drezewski, K. Obrocki, and L. Siwik. Agent-based co-operative co-evolutionary algorithms for multi-objective portfolio optimization. In A. Brabazon, M. ONeill, and D. Maringer, editors, *Natural Computing in Computational Finance*, volume 293 of *Studies in Computational Intelligence*, pages 63–84. Springer Berlin / Heidelberg, 2010.

[9] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[10] M.-R. Feizi-Derakhshi, H. Asil, and A. Asil. Proposing a new method for query processing adaption in database. *CoRR*, abs/1001.3494, 2010.

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[12] R. Ghaemi, A. Fard, H. Tabatabaee, and M. Sadeghizadeh. Evolutionary query optimization for heterogeneous distributed database systems. *World Academy of Science*, 43:43–49, 2008.

[13] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[14] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, Aug. 2008.

[15] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N relational joins. *ACM Transactions on Database Systems*, 9:482–502, 1984.

[16] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28:121–123, March 1996.

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

[18] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. *Sigmod Record*, 16:9–22, 1987.

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

[20] M. Matysiak. Efficient optimization of large join queries using tabu search. *Inf. Sci. Inf. Comput. Sci.*, 83(1-2):77–88, Mar. 1995.

[21] 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*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[22] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-m. Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *Proceedings of the International Database Engineering & Applications Symposium*, IDEAS '01, pages 20–33, Washington, DC, USA, 2001. IEEE Computer Society.

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

[24] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 367–376, New York, NY, USA, 1989. ACM.

[25] A. Swami and A. Gupta. Optimization of large join queries. *SIGMOD Rec.*, 17(3):8–17, June 1988.

[26] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 345 –354, apr 1993.

[27] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *Proceedings of the 4th international conference on extending database technology: Advances in database technology*, EDBT '94, pages 287–300, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[28] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. *SIGMOD Rec.*, 25(2):35–46, June 1996.

[29] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009.

[30] E. Zafarani, M. F. Derakhshi, H. Asil, and A. Asil. Presenting a new method for optimizing join queries processing in heterogeneous distributed databases. *International Workshop on Knowledge Discovery and Data Mining*, 0:379–382, 2010.