Expert Systems with Applications 41 (2014) 6934–6944

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Query join ordering optimization with evolutionary multi-agent systems

Frederico A.C.A. Gonçalves [a,c], Frederico G. Guimarães [a,*], Marcone J.F. Souza [b]

[a] Department of Electrical Engineering, Federal University of Minas Gerais, Belo Horizonte, Brazil
[b] Department of Computer Science, Federal University of Ouro Preto, Ouro Preto, Brazil
[c] IT Center, Federal University of Ouro Preto, Ouro Preto, Brazil

ARTICLE INFO

Article history:
Available online 17 May 2014

Keywords:
Join ordering problem
Query optimization
Multi-agent system
Evolutionary algorithm
Heuristics

ABSTRACT

This work presents an evolutionary multi-agent system applied to the query optimization phase of Relational Database Management Systems (RDBMS) in a non-distributed environment. The query optimization phase deals with a known problem called query join ordering, which has a direct impact on the performance of such systems. The proposed optimizer was programmed in the optimization core of the H2 Database Engine. The experimental section was designed according to a factorial design of fixed effects and the analysis based on the Permutations Test for an Analysis of Variance Design. The evaluation methodology is based on synthetic benchmarks and the tests are divided into three different experiments: calibration of the algorithm, validation with an exhaustive method and a general comparison with different database systems, namely Apache Derby, HSQLDB and PostgreSQL. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with lower cost plans and faster execution times in the majority of the cases.

1. Introduction

Database Management Systems (DBMS) are very complex software systems designed to define, manipulate, retrieve and manage data stored in a database. DBMS have an essential role in the information based society and represent critical components of business organization. Relational Database Management Systems (RDBMS) are those based on the relational model, which is the focus of this work. The information recovery depends on a database query and, as in Fig. 1, this query can be formed by many relations, which can be filtered and/or connected by different relational operators (Garcia-Molina, Ullman, & Widom, 2008) such as: selection ($\sigma_{\langle condition\rangle}$), projection ($\pi_{\langle attributes\rangle}$), intersection ($\bigcap$), union ($\bigcup$), set difference ($\setminus$), join ($\bowtie_{\langle condition\rangle}$) and others.

When processing a query, many steps can be executed by the RDBMS (Garcia-Molina et al., 2008; Elmasri & Navathe, 2010) until the delivery of a result: (i) scanning/parsing/validation, (ii) query optimization, (iii) query execution and (iv) result. The query optimization step, focus of this work, has a very important optimization task, which is ordering the relational operations of the query. Specifically in the case of the join operations, the most time-consuming operation in query processing (Elmasri & Navathe, 2010), the optimization task can be viewed as a combinatorial optimization problem commonly known as join ordering problem. The problem has similarities to the Traveling Salesman Problem (TSP) and according to Ibaraki and Kameda (1984), it belongs to the NP-Complete class. It is worth noting that after the optimization phase, the executor component receives a plan with all the instructions to its execution. Execution plans with relations ordered and accessed in a way that can cause high I/O and CPU cycles (high cost solutions) will impact directly in the query response time and affect the entire system. Besides, some costly plans can make the query execution impractical, because of their high execution times. Therefore, the use of techniques capable of finding good solutions in lower processing time is extremely important in a RDBMS. For instance, in one case reported in our experiments (Section 4) the proposed optimizer (Section 3.2) was capable to find a solution with a lower estimated cost, which allowed the plan to be executed almost 23% faster than the solution provided by the official optimizer used in H2 in the same experiment.

In this paper we propose an approach to query optimization that can be classified as a non-exhaustive one. We describe an evolutionary multi-agent system (EMAS) (see Section 3.1) for join ordering optimization running in the core of a real RDBMS named H2[1] in a non-distributed environment. The main feature of the

* Corresponding author. Tel.: +55 (31) 3409 3419.
E-mail addresses: fred@nti.ufop.br (F.A.C.A. Gonçalves), fredericoguimaraes@ufmg.br (F.G. Guimarães), marcone@iceb.ufop.br (M.J.F. Souza).

[1] http://www.h2database.com/.

http://dx.doi.org/10.1016/j.eswa.2014.05.005
0957-4174/© 2014 Elsevier Ltd. All rights reserved.

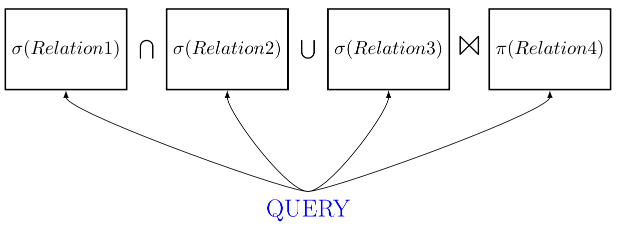


Fig. 1. Example query.

algorithm is having a team of intelligent agents working together in a cooperative or competitive way to achieve the solution of the problem. The agents of the system are able to interact and evolve in parallel. We extend initial ideas presented in Gonçalves, Guimarães, and Souza (2013) by including the following contributions: improvements in the algorithm operators; parallel implementation of the local search heuristics; evaluation of the parameters of the algorithm in the calibration phase; extended experiments with more realistic data; and comparison of the proposed algorithm with the official query planner in H2 and other DBMS, namely HSQLDB, Derby and PostgreSQL.

We highlight as the main contribution of this paper, the development of a technique not yet explored in the join ordering optimization field. Such method solves the join ordering problem in a parallel way, and as will be reviewed in Section 2, most of the proposed algorithms in the literature process the related problem sequentially. Still regarding the proposed algorithm, a new crossover method can be highlighted. The experiment design is validated with a real Database Management System, and consequently, a real cost model. Finally, we emphasize a more realistic evaluation methodology of the algorithms.

The paper is organized as follows. Section 2 discusses in more detail the query optimization problem and some methodologies applied to solve it. The proposed optimizer is detailed in Section 3. The evaluation methodology is introduced in Section 4. The computational experiments and the conclusions/future work are presented in Sections 5 and 6, respectively.

2. Query optimization problem

The optimization problem of this work consists in defining the best order of execution of the join operations among the relations specified in the query to be processed. According to Ioannidis (1996), the solution space can be divided into two modules: the algebraic space and the space of structures and methods. The algebraic space is the focus of the discussion in this section, because refers to the execution order of the relational operations considered. The space of structures and methods, on the other hand, is related to the available methods for relational operators in the RDBMS (more information about these methods can be found in Garcia-Molina et al. (2008) and Elmasri & Navathe (2010)).

The final result of the optimizer is a plan with all the necessary instructions to the query execution. Besides the operations order, a query can be represented by many different shapes. Assume that, for example, a join operator for 4 relations (*multi-way join*) is available and then a query with 4 relations could be expressed by only one join operation. However, in practice, the join operation is binary (*two-way join*) because the combinations for multi-way joins grow very rapidly (Elmasri & Navathe, 2010) and there are mature and proven efficient implementations for binary join in the literature. A representation commonly used by RDBMS is called left-deep-tree (see Fig. 2). This representation has only relations at the leaves and the internal nodes are relational operations. Due to this representation, the join operation is treated as binary (join

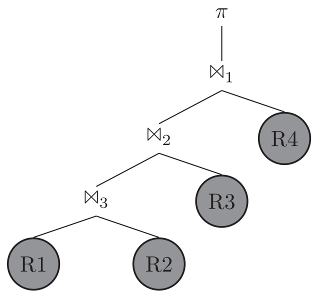


Fig. 2. Left-deep tree.

two tables only). Even with these restrictions, the number of possible solutions remains high – for a query with $N+1$ relations the number of solutions is given by $\binom{2N}{N}N!$. Further information about the join ordering problem can be found in Ibaraki and Kameda (1984), Swami and Gupta (1988), Ioannidis (1996) and Steinbrunn, Moerkotte, and Kemper (1997).

It is worth noting that the cost of a solution is not given necessarily by the actual cost of executing the query, but instead, can be given by a cost function F that estimates the real cost of the solution. The cost estimation can use many metrics, for instance: I/O, CPU and Network. The optimizer relies on the RDBMS for the task of estimating the cost of the solution. Therefore, in the optimization process, this function is abstracted and treated as a black box, just receiving a candidate solution and returning its (estimated) cost. Information about the cost model of relational operations are provided by Garcia-Molina et al. (2008) and Elmasri and Navathe (2010).

Listing 1 presents a practical example with a query that returns all marks from computer science students.

Two join operations can be extracted from the previous query: $J_1 = \{student \bowtie marks\}$ and $J_2 = \{student \bowtie dept\}$. Besides, two valid solutions can be identified in this simple example: $S_1 = \{J_1, J_2\}$ and $S_2 = \{J_2, J_1\}$, one with a lower cost than the other. In practice, the problem can have a huge combination of possible solutions, preventing the use of exact methods or exhaustive approaches. Nonetheless, non-exhaustive algorithms fit well in these situations.

Given the complexity of the problem, several studies were presented for non-distributed environments along the years since the early days of relational databases in the 1970s (Codd, 1970). The seminal work is presented by Selinger, Astrahan, Chamberlin, Lorie, and Price (1979), advancing an exhaustive method based on dynamic programming (*DP*) with a time complexity $O(N!)$, where N stands for the number of relations in the query. Ibaraki and Kameda (1984) presented two algorithms, named A and B, with a time complexity $O(N^3)$ and $O(N^2 \log N)$, respectively. An extension of Algorithm B named *KBZ* algorithm with time complexity $O(N^2)$ is presented by Krishnamurthy, Boral, and Zaniolo (1986). A simulated annealing (*SA*) version for the current problem is presented in Ioannidis and Wong (1987). Many methods are compared in Swami and Gupta (1988): Perturbation Walk, Quasi-random Sampling, Iterated Improvement (*II*), Sequence Heuristic and SA. Additionally, in Swami and Gupta (1988), the authors created a new evaluation methodology to check the heuristics,

```
SELECT name, disc, mark FROM student,
    marks, dept WHERE id_student=id_marks
    AND dpt_student=id_dept AND id_dept='
    DECOM'
```

Listing 1. Example – query SQL.

considering cardinality, selectivity of the join predicates, distinct values and indexes. Based on the methods *II* and *SA*, an algorithm called *2PO* has been proposed in Ioannidis and Kang (1990), which combines the two techniques mentioned. A genetic algorithm (GA) applied to the join ordering problem is presented in Bennett, Ferris, and Ioannidis (1991) and compared against the *DP* (Selinger et al., 1979). The authors in Swami and Iyer (1993) proposed an extension of the algorithm (*AB*) with some improvements and a time complexity $O(N^4)$. The meta-heuristic Tabu Search (*TS*) (Glover & Laguna, 1997) is explored for this problem in Matysiak (1995). An algorithm named *blitzsplit* is presented by Vance and Maier (1996). Another comparison between many algorithms is presented in Steinbrunn et al. (1997). SA and GA algorithms were applied again in Lee, sheng Shih, and huei Chen (2001) and Dong and Liang (2007), respectively. In the work by Guttoski, Sunye, and Silva (2007), an implementation of the *kruskal* algorithm is described. A parallel based *DP* algorithm is discussed in Han, Kwak, Lee, Lohman, and Markl (2008). Many methods in the literature are reviewed by Lange (2010) in his Master thesis.

More recently, the GPU technology in database systems was addressed by Heimel and Markl (2012) and Heimel (2013). They discuss about some initial ideas for the design of a GPU-assisted query optimizer, with plans to implement a GPU-accelerated version of the DP algorithm (Selinger et al., 1979). The generation and use of hybrid query plans, i.e., the optimization and execution of queries with plans that mix the use of CPU and GPU processors is discussed in Breß, Schallehn, and Geist (2013). Regarding distributed databases, the authors in Sevinç and Coşar (2011) have proposed a new GA version (NGA) and compared it against a previous GA method (Rho & March, 1997), exhaustive and random algorithms. According to the authors, the NGA was capable to find optimal results in 80% of cases and improved the results over previous GA in 50%. A review about some algorithms applied to query optimization in distributed database systems can be found in Tewari (2013). Lastly, in Golshanara, Rouhani Rankoohi, and Shah-Hosseini (2014), an ant-colony optimizer (ACO) (Dorigo, Maniezzo, & Colorni, 1996) is proposed to order the join operations of an environment with the data replicated across multiple database sites. The ACO is compared with other algorithms, among them, a GA method (Sevinç & Coşar, 2011). According to the results, the ACO reduced the optimization time in about 80% without lost quality of the solutions.

The first use of GA and multi-agent systems (MAS) for query optimization in distributed DBMS is proposed in Ghaemi, Fard, Tabatabaee, and Sadeghizadeh (2008). They define the following agents: Query Distributor Agent (QDA) to divide the query into sub-queries, Local Optimizer Agents (LOAs) that applies a local genetic algorithm and Global Optimizer Agent (GOA) responsible for finding the best join order between the sites. In a comparison with a dynamic programming method, the authors verified that their approach takes less time for processing the queries. An extension of Ghaemi et al. (2008) with focus on building an adaptive system is given by Zafarani, Derakhshi, Asil, and Asil (2010) and Feizi-Derakhshi, Asil, and Asil (2010). The results have shown a reduction of up to 29% in time response. It is worth noting the distinctions between the present work and the work from Ghaemi et al. (2008), Zafarani et al. (2010) and Feizi-Derakhshi et al. (2010). First, their methods are supposed to run in a distributed environment; secondly, they are running outside of the DBMS; and lastly, the agents have a limited and different way of interaction. Basically, one agent (QDA) breaks the query into pieces and distributes part of the analysis of relations of some data source with a registered agent (LOA) that will execute a standalone version of genetic algorithm to find a possible order to join the associated relations. Finally, the last agent (GOA) will try to minimize the network traffic, by executing the partial plan defined by the related LOA, sending the partial result across the network to another data source, and joining it with other partial results of plans defined by other LOA. The process terminates when all partial results are joined.

The join ordering problem is not restricted only to traditional database systems, in the Resource Description Framework (RDF) field (Klyne & Carroll, 2014), a typical scenario is composed by many possible interconnected heterogeneous sources of data distributed over network. The join ordering problem arises when a RDF query (Prud'hommeaux & Seaborne, 2008) needs to join many sources of data. The join operation in this context is similar to the relational databases and has impact on the response time. Hogenboom, Frasincar, and Kaymak (2013) have compared an ACO approach against an 2PO (Stuckenschmidt, Vdovjak, Broekstra, & Houben, 2005) and GA (Hogenboom, Milea, Frasincar, & Kaymak, 2009) methods in the join ordering of the sources. The results of ACO have shown lower execution times and better solutions quality for queries consisting of up to 15 joins. For larger problems the GA performed better.

In relation to some RDBMS in the market, *H2*[2] uses a brute force method for queries with up to 7 relations and a mixed algorithm composed by an exhaustive, greedy and random search methods is applied to queries with more than 7 relations. The PostgreSQL[3] has an optimizer based on DP (Selinger et al., 1979) for ordination of queries with up to 11 relations and a GA for queries with more than 11 relations. The RDBMS MySQL[4] and Apache Derby[5] employ a depth-first search based algorithm.

3. Query optimizer

This section discusses the proposed optimizer applied to the join ordering problem presented previously. Section 3.1 introduces concepts about evolutionary multi-agent systems (EMAS) and some of their applications. The proposed evolutionary multi-agent optimizer is explained in Section 3.2.

3.1. Evolutionary multi-agent systems – EMAS

In this subsection we provide an overview of the use of multi-agent systems and evolutionary algorithms in optimization. Multi-agent systems can be defined as systems involving teams of autonomous agents working together in a cooperative or competitive way to achieve the solution of a given problem. Such systems differ from purely parallel systems, because of the distinctive interaction between the agents. The related agents have specific characteristics such as reactivity, proactivity, sociability and so on (Wooldridge, 2009). Evolutionary algorithms, on the other hand, work with a population of candidate solutions and 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.

More recently, some researchers have explored the integration between evolutionary algorithms and multi-agent systems, trying to take the best from both worlds. The integration between multi-agent systems and evolutionary algorithms lead to the so-called evolutionary multi-agent systems – EMAS ('t Hoen et al., 2004; Hanna & Cagan, 2009; Barbati, Bruno, & Genovese, 2012).

[2] H2 – Version 1.3.174 – 2013 www.h2database.com/html/performance.html.
[3] PostgreSQL – Version 9.2.6 – 2013 www.postgresql.org/docs/9.2/interactive/geqo.html.
[4] MySQL – Version 5.1.72 – 2013 http://dev.mysql.com/doc/internals/en/optimizer.html.
[5] Derby – Version 10.9.1.0 – 2012 http://db.apache.org/derby/papers/optimizer.html.

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 (Drezewski, Obrocki, & Siwik, 2010).

Evolutionary multi-agent systems have been applied in different contexts, among them: decision making (Dahal, Almejalli, & Hossain, 2013; Khosravifar et al., 2013), multi-agent learning (Enembreck & Barthès, 2013; Van Moffaert et al., 2014; Li, Ding, & Liu, 2014), multi-objective optimization (Drezewski et al., 2010; Tao, Laili, Zhang, Zhang, & Nee, 2014).

3.2. Proposed optimizer

In this work we present a method based on evolutionary algorithms and multi-agent systems. It employs techniques inspired by evolutionary models and is composed by a set of agents that work together to achieve the best possible solution for the problem, i.e., the best join order for the relations in the query. Such methodology extends the default behavior of genetic algorithms and multi-agent systems, since the agents can act pro-actively and reactively. They can adopt specific interaction mechanisms and explore the solution space in a smart way by applying genetic operators and constructive heuristics. Drezewski et al. (2010) cites two different evolution mechanisms: mutual selection and host-parasite interaction. The mutual selection was the technique adopted in this work, in which each agent chooses individually another agent and executes the available and related actions.

The proposed evolutionary multi-agent system can be described as $EMAS = \langle E, \Gamma, \Omega \rangle$, where E is the environment, Γ is the system resources set and Ω is the information available to the agents in the system.

The environment is non-deterministic and dynamic, there is no certainty about the results of an agent action and agents can change the environment. The environment definition is given by $E = \langle T^E, \Gamma^E, \Omega^E \rangle$, wherein T^E represents the environment topography, Γ^E the resources of the environment and Ω^E the information available to the agents. 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 corresponds to Ω^E.

The topography is represented by the notation $T^E = \langle A, l \rangle$, where A is the set of agents (population) in the environment and l is a function that allows to locate a specific agent. Each agent $a \in A$ can be denoted by the expression $a = \langle sol^a, Z^a, \Gamma^a, PR^a \rangle$, sol^a stands for the solution represented by the agent, which is an integer array and each element of this array represents a relation id. This array also defines the execution order to join the relations. The actions set of the agent is given by Z^a and its life points by Γ^a. The current goal of the agent is defined by its current profile PR^a. The initial life of all agents is expressed in terms of $LIFE = IL \times NR$, where IL is the initial life coefficient and NR is the number of relations in the query. All actions available in the environment are listed as follows:

- ***getLife*** (***gl***) – Used to obtain life points from another agent;
- ***giveLife*** (***vl***) – Give life points to another agent;
- ***lookWorse*** (***lw***) – Search for an agent with worse fitness value;
- ***lookPartner*** (***lp***) – Search for a partner in the set of agents in the environment;
- ***crossover*** (***cr***) – Crossover operator used to generate offspring;
- ***mutation*** (***mt***) – Mutation operator;
- ***becomeBestSol*** (***bb***) – Update its solution with the best current solution;
- ***updateBestSol*** (***ub***) – Update the best solution;
- ***randomDescent*** (***rd***) – Random descent local search method;
- ***parallelCompleteDescent*** (***pd***) – Parallel Best Improvement local search method;
- ***semiGreedyBuild*** (***gb***) – Construct a solution by using a semi-greedy heuristic;
- ***processRequests*** (***pr***) – Evaluate and process pending requests;
- ***tryChangeProfile*** (***cp***) – Tries to change the profile of the agent;
- ***stop*** (***st***) – Stops momentarily its execution;
- ***die*** (***id***) – Action of dying, when the life points of the agent ends.

Each agent is associated to 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 mutation and local search methods employ swap and reallocation movements (Fig. 3).

By default, the mutation can apply in each execution at most 5 movements (swap or reallocation) and the random descent method has a maximum number of iterations without improvement equal to $RD_{mov} = RDE \times IL$, where RDE is a coefficient of effort and IL the initial life of an agent.

Three crossover operators were implemented: *Ordered Crossover* – OX (Davis, 1985), *Sequential Constructive Crossover* – SCX (Ahmed, 2010) and a new operator proposed in this work named *Pandora-Aquiles Greedy Crossover* – PAGX. The examples described next use as input the information of the Fig. 4.

In the crossover OX, two descendants can be generated. First a crossover point is selected at random, which defines the part of one of the agents will go to the descendant. The rest of the sequence is taken from the other agent in an ordered way avoiding repetition of elements. Considering the agents in Fig. 4 and a crossover point comprising the 1st e 2nd relations of Agent 1, one of the resulting descendants is presented in Fig. 5.

The strategy SCX starts by adding the first relation R_{CUR} of the Agent 1 in the resulting descendant. After that, the next relation R_{NEXT} subsequent to R_{CUR} from one of the parents, not present in the descendant and with lower join cost $R_{CUR} \bowtie R_{NEXT}$ is chosen. Then R_{NEXT} becomes R_{CUR} and the process continues until all relations are added in the descendant as in Fig. 6. The cost of the join operations between each pair of relations is stored in a cost matrix Fig. 4(b) and ties are solved randomly.

The PAGX method mixes randomness and determinism. Initially a crossover point is chosen as in OX. However, the resulting descendant inherits the partial solution with lower cost from one of the parents or from the first agent in the case of a tie. After that, a list L with the remaining relations is created and in each iteration, the relation with lower join cost is taken from L or the first element in L if a tie occurs. The process ends when L becomes empty. An example result considering a cut point with the 2nd e 3rd relations and the agents from Fig. 4(a) is presented in Fig. 7.

The action semiGreedyBuild is based on a semi-greedy (greedy-random) heuristic. At each iteration, a list CL containing all relations not present in the current solution is sorted according

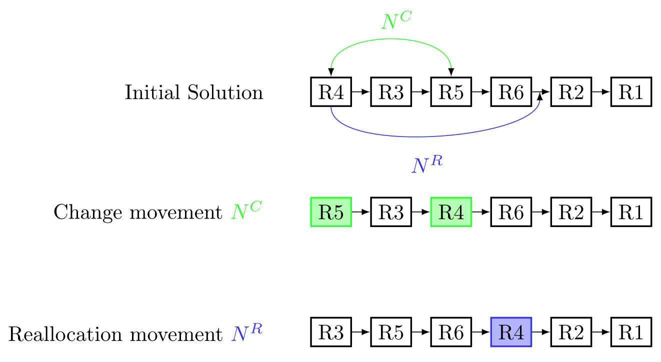
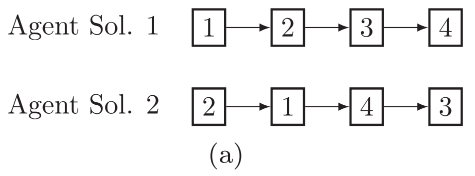


Fig. 3. Movement types.



Join 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

(b)

Fig. 4. Agents informations.

to a greedy function or classification criterion. The best ranked elements from *CL* form the restricted candidates list *RCL*. A new relation is taken randomly from the *RCL* and inserted in the current solution. The method stops when all relations have been inserted. In this work, we adopt two different classification criteria to *CL*: the *ERX* criterion that takes into consideration the number of links between each relation and relations with less links are better classified and the *SCX* criterion that is based on the cost of links or on the join cost of the relations, where relations with lower join costs are better classified.

Lastly, the best current solution is refined by the parallel best improvement method (*parallelCompleteDescent*) and a local optimum according to the neighborhood N^C or N^R (selected at random) will be returned. The method has a task list *TL* shared by the agents, which distributes all the neighborhood analysis into distinct tasks. Thus, each task evaluated by the agents will be taken from *TL*. The process ends when all tasks have been evaluated. It is worth noting that each task will be analyzed by only one agent and no more than one time. The method returns the best solution of one of the agents.

Each agent can execute different goals according to specific profiles, they can generate a solution together with another agent, refine its own solution and so on. The designed profiles 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 and tries to explore other parts of the solution space;
- **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.

Regarding the defined profiles, it is worth mentioning that to avoid high randomness and high effort of refinement, the probabilities of choosing the MUTANT and RANDOM_DESCENT

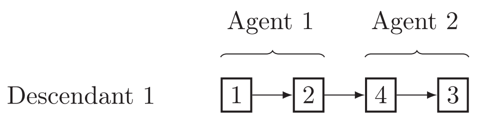


Fig. 5. Descendant OX.

profiles were set as 10% for each one. On the other hand, to a smart and effective exploration of the solution space, the REPRODUCTION and SEMI-GREEDY profile have received a probability equal to 60% AND 15%, respectively. Finally, in order to impose some selective pressure to the algorithm, the probability of RESOURCE profile was set as 5%. The set of actions of each profile is distributed as follows:

- **RESOURCE** – lookWorse, processRequests, getLife, tryChangeProfile, stop and die;
- **REPRODUCTION** – lookPartner, processRequests, crossover, updateBestSol, tryChangeProfile, stop and die;
- **MUTANT** – mutation, processRequests, updateBestSol, tryChangeProfile, stop and die;
- **RANDOM_DESCENT** – becomeBestSol, randomDescent, processRequests, updateBestSol, tryChangeProfile, stop and die;
- **SEMI-GREEDY** – semiGreedyBuild, processRequests, updateBestSol, tryChangeProfile, stop and die.

The pseudo-code of the Evolutionary Multi-Agent Query Optimizer (MAQO) is presented in Algorithm 1. This algorithm first initializes the system (line 2). Then, all agents are created with the profile *REPRODUCER*. One of them (line 4) is started with an initial solution in the order in which the relations are read during the initial parsing of the query and the other ones are filled with a solution generated by the semi-greedy heuristic. The initialization and waiting of the agents and final parallel refinement of the best current solution are done in lines 9, 11 and 20.

Algorithm 1. MAQO

```
   Data: QueryRelations
   Result: $sol^*$
 1 begin
 2     Environment $E \leftarrow$ initializeEnvironment(QueryRelations);
 3     //Create the agents with the profile REPRODUCTION;
 4     createAgentWithInitialSolution($E$, REPRODUCTION);
 5     for $i$=2 to $E \rightarrow MAXAGENTS$ do
 6         createAgentWithSemiGreedySolution($E$, REPRODUCTION);
 7     end
 8     //Initialize threads;
 9     initializeAgents($E$);
10     //Wait threads finish;
11     waitAgents($E$);
12     //All agents update their solutions;
13     for $i$=1 to $E \rightarrow MAXAGENTS$ do
14         $E \rightarrow$findAgent(i)$\rightarrow$becomeBestSol();
15     end
16     //Generate the tasks that will be executed by the agents;
17     List $TASKS \leftarrow$ generateTasks();
18     //Parallel Best Improvement in the best current solution;
19     for $i$=1 to $E \rightarrow MAXAGENTS$ do
20         $E \rightarrow$findAgent(i)$\rightarrow$parallelCompleteDescent($TASKS$);
21     end
22     return $E \rightarrow bestSolution$;
23 end
```

Once the agents have been initialized, the evolutionary process will also start, since agents may evolve through the execution of the associated actions, which can be individual or related to

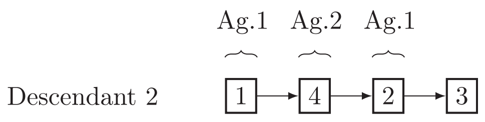


Fig. 6. Descendant SCX.

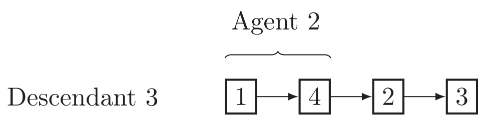


Fig. 7. Descendant PAGX.

another agent. At every iteration of the agent in a given profile, its life points decrease. The agents can change their profiles only after a minimal number of iterations in the current profile. They will always change to *RESOURCE* profile when the life level reaches 10% of the initial life. Those agents with no more life points execute the action *stop*. When all agents stop, the optimizer executes the parallel best improvement action and then finally all agents die (action *die*) and the algorithm terminates its execution.

4. Evaluation methodology

The evaluation methodology of the problem (Section 2) in this work involves the optimization and execution of a group of queries in SQL. Such process can be based on synthetic benchmarks built according to some criteria as done in Swami and Gupta (1988), Swami (1989), Vance and Maier (1996), Steinbrunn et al. (1997), Brunie and Kosch (1997), Lee et al. (2001), Shapiro et al. (2001), Dong and Liang (2007), Lange (2010) or on those benchmarks standardized by the industry, such as the TPC-DS (decision support benchmark) from TPC (*Transaction Processing Performance Council*).[6] The TPC-DS as well as other benchmarks from TPC were discarded in our experiments, because our minimal number of relations in the queries starts with 12, and for example the TPC-DS has only 7 out of 99 queries with 8 or more relations.

The methodology proposed in this work is based mainly on the ideas presented by Swami and Gupta (1988) and Swami (1989). We randomly create the test database and the test queries according to some criteria. In an attempt to better approximate real problems, some of the database construction criteria were based on observed distributions in a production database from the IT Center of a Brazilian University. The evaluation methodology is divided in two stages, one builds the database and another builds the queries. To create and load the database, the following criteria were used:

- **Cardinality distribution of tuples** – $[10, 100)$ – 26%, $[100, 1000)$ – 27%, $[1000, 1000000]$ – 47%;
- **Distribution of distinct values of the tuples** – $[0, 0.2)$ – 30%, $[0.2, 1)$ – 16%, 1 – 54%;
- **Relations Columns** – three per relation, being one reserved for primary key;
- **Probability of a column to have indexes and foreign keys** – 90% and 20%, respectively.

The total number of relations created and loaded according to the previous distributions was set as 60. The cardinality and the number of distinct values of the columns were selected at random from the given intervals. The evaluation of complex queries with many relations fits well in decision support problems and data mining (Brunie & Kosch, 1997; Dong & Liang, 2007). There are problems in the literature with up 100 relations. The number of relations presented in the queries was fixed in: 12, 16, 20, 30, 40 and 50 relations. Besides, the generation of the test queries was guided by three different graph shapes: chain, star and snowflake.

The generation of a query in a specific shape, first selects at random the given number of relations from the 60 available. After that, the join predicates are formed by randomly selecting the columns from both relations. It is important to note that all projected columns are the same of the join clauses and the predicates always will be generated according to the graph shape. Besides, multiple seeds can be used to generate different queries and an example of a query built with shape chain and with the relations $R_1, R_7, R_2, R_{10}, R_{60}$ and R_{50} is presented in Listing 2.

$$\begin{aligned}&\textbf{SELECT} * \textbf{FROM}\ R_1, R_7, R_2, R_{10}, R_{60}, R_{50}\ \textbf{WHERE}\\&\quad R_1.col_1 = R_7.col_2\ \textbf{AND}\ R_7.col_3 = R_2.col_2\ \textbf{AND}\\&\quad R_2.col_1 = R_{10}.col_3\ \textbf{AND}\ R_{10}.col_1 = R_{60}.col_3\ \textbf{AND}\\&\quad R_{60}.col_2 = R_{50}.col_1\end{aligned}$$

Listing 2. Example – SQL chain scheme.

Table 1
Factors and levels – calibration experiment.

Factor	Level
NA	8, 16
IL	1, 1.5
DL	0.03, 0.05, 0.1
RDE	0.3, 0.5
MM	Change, Reallocation
RDM	Change, Reallocation
CM	PAGX, SCX, OX
SM	ERX, SCX

Table 2
Significance of the factors and their interactions.

Factor	Significance		Interaction Signif.	
	Cost	Setup	Cost	Setup
NA	Ye	Yes	MM	MM, SM, RDM,IL and DL
IL	Yes	Yes	–	DL
DL	Yes	Yes	–	–
RDE	No	No	–	–
MM	No	No	NA	NA
RDM	Yes	Yes	DL	IL, DL
CM	Yes	Yes	SM, RDM, RDE, IL	RDM, IL
SM	Yes	Yes	RDM, IL	DL

Table 3
ANOVA – efficiency comparison.

Source	DF	SS	MS	Iter	Pr
DB	1	7e+28	7e+28	51	0.961
SHAPE	1	1e+33	1e+33	1009	0.090.
DB:SHAPE	1	2e+26	2e+26	51	0.961
SIZE	1	2e+34	2e+34	5000	<2e–16***
DB:SIZE	1	1e+29	1e+29	51	0.725
SHP:SIZE	1	2e+33	2e+33	2906	0.033*
DB:SHAPE:SIZE	1	3e+26	3e+26	51	0.863
Residuals	16	8e+33	5e+32		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1.

5. Experiments

The experimental section is divided in three subsections: the first (5.1) for the calibration of the algorithm parameters, the

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

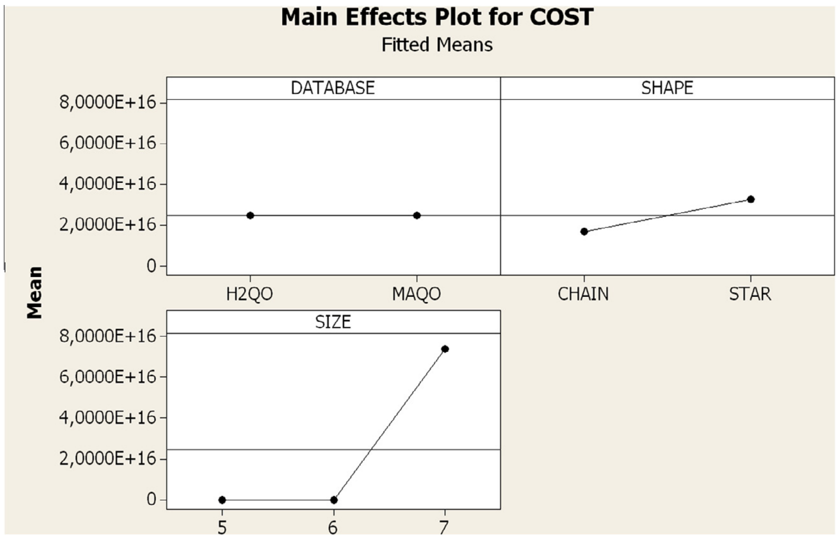


Fig. 8. Main effects – cost efficiency.

Table 4
Factors and levels – general experiment.

Factor	Level
DATABASE	H2QO, MAQO, HSQLDB, DERBY, POST
SHAPE	Chain, Star, Snowflake
SIZE	12, 16, 20, 30, 40 and 50

Table 5
ANOVA – general experiment.

Source	DF	SS	MS	Iter	Pr
DB	4	6e+12	1e+12	5000	<2e−16***
SHAPE	2	5e+11	2e+11	5000	0.0008***
DB:SHAPE	8	3e+12	4e+11	5000	<2e−16***
SIZE	1	6e+11	6e+11	5000	<2e−16***
DB:SIZE	4	3e+12	8e+11	5000	<2e−16***
SHAPE:SIZE	2	3e+11	1e+11	3606	0.0541·
DB:SHAPE: SIZE	8	2e+12	3e+11	5000	<2e−16***
Residuals	150	9.04e+12	6.03e+10		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '·' 0.1 ' ' 1.

second (5.2) to test the efficiency of the MAQO in finding global optima and the last (5.3) to compare the RDBMS H2 running the MAQO against other RDBMS. It is noteworthy that all experiments have used a factorial design of fixed effects, the statistical analysis used the R/Minitab softwares and was done with Permutational ANOVA (Permutations Test for an Analysis of Variance Design by Anderson & Braak (2003)) and all the related assumptions were not violated by the generated models. The confidence interval for the experiments was set as 95%, implying a significance level $\alpha = 0.05$ and all observations were randomized.

For each ANOVA table hereafter, DF represents the *Degrees of Freedom*, SS represents the *Sum of Squares*, MS means *Mean Square*, Iter is the number of iterations until the criterion is met and Pr refers to the *p*-value of the related statistical test.

The queries were executed in a machine with *Core i7-2600 CPU 3.40* GHz, with 16 GB *RAM*, operating system *Ubuntu* 10.10-x86_64. The timeout for the execution of the queries was set as 2 h and the maximum number of resulting tuples was limited in 10 million. According to the H2 database, a query with 50 relations and 50 projected columns could generate a result with about 9 GB for 10 million tuples. We adopted the strategy of cold cache in all executions. Thus, before the execution of each query, the OS pages in the memory are synchronized and then freed. Each one of the experiments have used different seeds to generate the distinct queries.

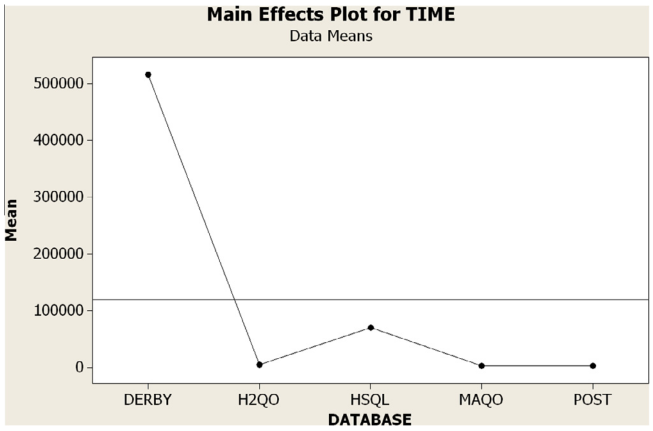


Fig. 9. Main factor – DATABASE.

Table 6
Best execution times overview.

	H2QO	HSQL	DERBY	POSTGRESQL
MAQO	74.78%	72.17%	100%	63.3%

Finally, the RDBMS chosen to implement the optimizer was the H2 Database Engine and consequently, the cost model adopted is the same defined by the H2. For this reason, we considered only the method Nested-Loop-Join. We remark that the RDBMS 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.

5.1. Calibration of the algorithm

In order to calibrate the parameters of the optimizer, we have selected 1 test query with 40 relations in the chain and star graph shapes. The factors analyzed are: the maximum number of agents

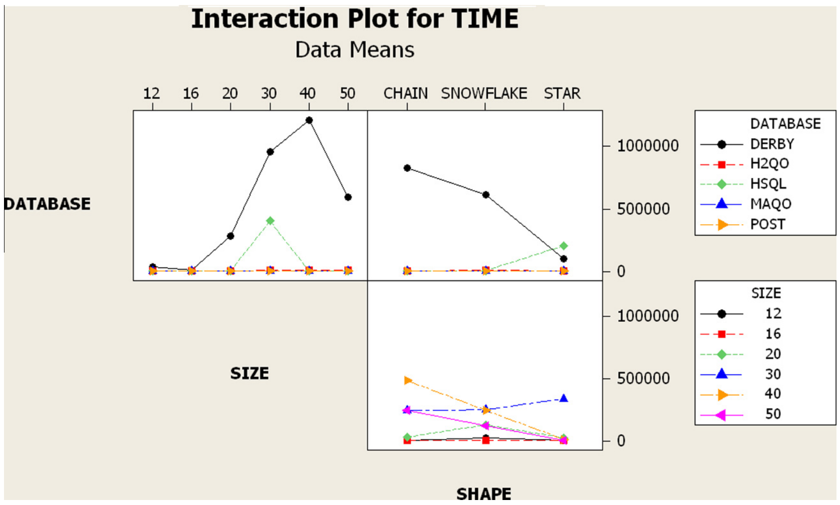


Fig. 10. Main effects and interactions – general experiment.

(NA), mutation type movement (MM), crossover method (CM), random descent type movement (RDM), semi-greedy method (SM), initial life coefficient (IL), decrement life coefficient (DL) and random descent effort (RDE). The initial life of the agents is given by the related coefficient times the number of relations in the query, the life decremented in each agent iteration is expressed by the initial life times the related coefficient and the number of iterations of the random descent without improvement is given by the effort times the number of relations in query. Table 1 describes the levels for each factor.

We have considered two complete replications, totaling 2304 observations. The response variables are the cost of the generated plans and the setup time in milliseconds. The test hypotheses are no effect for the main factors and their interactions for all levels and significance for the main factors and/or their interactions for at least one of the levels. The parameters were selected by combining the effect of both response variables, trying to get lower costs but not with necessarily higher setup times. According to the ANOVA results, the factors with significance are distributed as follows and the related null hypotheses were rejected.

According to Table 2 and the related mean values of the factors and their interactions, the maximum number of agents was set as 8, probability of 50% to choose one of the movement types for both mutation and random descent methods, crossover method with 50% of chance to use PAGX or OX, semi-greedy method defined with SCX strategy and the values 1.5, 0.1 and 0.3 for initial life coefficient, decrement life coefficient and random descent effort, respectively.

5.2. Comparison with the exhaustive method

The goal of this Subsection is to present the potential of MAQO to find global optima. To accomplish that, the proposed optimizer was compared against the official brute force method H2QO of H2 database. The comparison have used 30 different queries for problems with 5, 6 and 7 relations and chain and star graph shapes, totaling 360 queries for each optimizer distributed into two complete replications. The main factors studied are: the database algorithm (MAQO and H2QO), the problem size (5, 6 and 7) and the query shape (chain and star). The test hypotheses are no effect for the factors/interactions for all levels and significance for at least one of the levels in the factors/interactions. The results related to the response variable cost are summarized in Table 3.

According to the ANOVA Table, only the factor SIZE and its interaction with SHAPE have significance, which allows us to reject the related null hypotheses and say that part of the variability of the data can be explained by them. Thus, no statistical significance was detected between the results of the algorithms. The MAQO was capable to find the optima in 90% of the cases. Fig. 8 shows graphically the mean costs of the main factors and its interactions.

5.3. Comparison with other database systems

This Section is intended to compute the total time spent in optimization and execution of a set of queries by 4 different RDBMS. We choose 3 databases implemented in Java language and one native system implemented in C language, they are: H2,[7] HSQLDB,[8] Derby[9] and PostgreSQL,[10] respectively. A total of 2 complete experimental replications have been considered and the test-set is composed by 3 different queries for problems with 12, 16, 20, 30, 40 and 50 relations and the shapes distributed in chain, star and snowflake, totaling 540 queries. The H2 database was tested with two different optimizers, one with the official algorithm described in Section 2 and another with the method proposed in this work, namely H2QO and MAQO, respectively. The mains factors and its levels are presented in Table 4.

Considering the total time as response variable in milliseconds, the ANOVA Table 5 shows that the main factors (and some of their interactions) have significance on data variability, allowing us to reject the related null hypotheses with 95% degree of confidence. The difference in the execution times is caused by the database systems, shape and the size of the queries.

Based on the main objective of this work, that is to compare the proposed algorithm running in a real database against different real systems, the factor DATABASE is the focus of the analysis in this subsection. Fig. 9 shows that the average execution time of the DBMS Derby is the higher and the MAQO and PostgreSQL are the lowest. These results show the superiority of the developed algorithm in relation to the official optimizer of the H2 database. Table 6 gives an overview of the queries executed with lower times against the other databases.

[7] H2 – Version 1.3.174, www.h2database.com.
[8] HSQLDB – Version 2.0.0, http://hsqldb.org/.
[9] Derby – Version 10.9.1.0, http://db.apache.org/derby.
[10] PostgreSQL – Version 9.2.6 – www.postgresql.org.

Table 7
Best plans overview.

Query shape	MAQO(%) – H2QO(%)					
	12	16	20	30	40	50
Chain	50-50	0-100	50-50	100-0	50-50	100-0
Star	0-100	100-0	100-0	100-0	100-0	100-0
Snowflake	50-50	100-0	100-0	100-0	100-0	100-0

Fig. 10 presents graphic information about the main factors and their interactions. According to the average time results of all database systems, the shape snowflake and size 30 consumed more time in the optimization and execution of the queries. On the other hand, the MAQO spent on average more time on queries in the shape Star and Snowflake and size equal to 50.

Finally, Table 7 summarizes the results obtained by the H2QO and MAQO algorithms in relation to the cost of the plans. It is clear that the evolutionary multi-agent algorithm performed better in the majority of cases. The H2QO overcomes the MAQO only in the problems Chain-16 and Star-12. Additionally, out of 108 executed queries by each optimizer, only 14% of plans generated by the default planner in H2 were better than the ones generated by MAQO.

6. Conclusions and future work

This paper presented a novel database query optimizer based on evolutionary algorithms and multi-agent systems and an evolution mechanism supported by mutual selection. Many actions and profiles have been designed to provide different goals to the agents and to explore the solution space of the query join ordering problem in a smart way. The algorithm employs classical genetic operators, constructive heuristics and refinement methods according to the associated agent profiles. We proposed a new crossover method called *Pandora-Aquiles Greedy Crossover*.

The execution environment can be described as accessible, because the agents can obtain precise, updated and complete informations about the environment. Besides, it can be characterized as deterministic and dynamic. The agents are hybrids: reactive with internal state. The following features of the agents can be cited: reactivity, proactivity, sociability, veracity and benevolence.

The evaluation methodology of this work follows mainly the ideas from Swami and Gupta (1988) and Swami (1989). We developed a benchmark according to distributions based on a real production database of an IT Center of a Brazilian University. The experiments were subdivided into 3 separated evaluations: calibration of the algorithm, validation of the algorithm with a brute force method and a comparison with 3 different database systems. A factorial design was applied in all experiments with a confidence interval set as 95%. Moreover, all the related assumptions were not violated by the generated models.

For the calibration phase, it was used one test query with 40 relations in the chain and star graph shapes. According to the results, the maximum number of agents was set as 8, probability of 50% to choose one of the movement types for mutation and random descent methods, crossover method with 50% of chance to use PAGX or OX, semi-greedy method defined with SCX strategy and the values 1.5, 0.1 and 0.3 for initial life coefficient, decrement life coefficient and random descent effort, respectively.

In the comparison of the proposed optimizer MAQO against the official H2 brute force method, we have used 30 different queries for problems with 5, 6 and 7 relations and chain and star graph shapes. The results showed that MAQO was capable to find the optima in 90% of the cases.

The general test has adopted 3 different queries for problems with 12, 16, 20, 30, 40 and 50 relations and the shapes: chain, star and snowflake. The proposed optimizer was compared against the official H2 official optimizer and the database systems HSQL, Derby and PostgreSQL. According to the results, the developed algorithm showed execution times better than the official planner H2QO in 74.78% of the cases, better than PostgreSQL in 63.3% of the cases and in 100% of cases than Derby. Besides, in the same experiment, compared with H2QO, the MAQO returned plans with lower costs in 86% of the cases.

Sections 1 and 2 show the importance of the join ordering problem. It is important to say that, good execution plans, in fact, lead to faster response times in the system. Considering that the synthetic database is based on real distributions and the queries are based on common problem representations, one can expect that in an environment with few users (low concurrency), as the results suggest, the use of the proposed algorithm over the official H2 optimizer would improve the response times in 74.78% in possible real world cases. Besides, the new optimizer allowed H2 to be better than the very popular RDBMS PostgreSQL in 63.3% of the problems. Finally, according to the superiority of H2 with the new planner in relation to Derby, in a real situation similar to the proposed synthetic environment, the best decision is choosing H2.

Therefore, in view of the superiority of this novel proposed approach over others in the literature, we show the benefits of using an evolutionary multi-agent system for query join ordering optimization. Moreover, the set of actions, the available profiles and interaction mechanism have proved to be an efficient tool to explore the solution space of the related problem and the parallel feature of the algorithm has allowed it to prepare the plans with lower setup times. As a possible weakness of the proposed optimizer, it is worthy to note that all experiments were executed with only a single caller, i.e., only one query was optimized per execution. Thus, in the calibration phase, the number of agents was configured as equal to the total number of available CPU cores. However, in a environment with more then one caller, such configuration could overload the CPU.

As future work, it is worthy to cite an extension of the experiments in order to run with more than one caller. This is a interesting point and could be used to explore the proposed optimizer in a environment with high load, which can represent many real world situations. Another very important future study is the execution of the optimizer in a distributed environment. As pointed out in the review presented in Section 2, distributed database systems is a field recently explored. Consequently, the application of the optimizer to generate plans in such environment would be another contribution. Moreover, EMAS are naturally indicated for and compliant with distributed environments and problems. Thus, with new actions and ways of interaction between the agents, the optimizer could generate plans in a distributed environment. Another area recently studied is related to the use of GPU processors to support some database components and its operations. Since the proposed algorithm has agents running in parallel, the use of GPU to support its execution could be another avenue to explore. At last, the similarities between the join ordering problem in the RDBMS and RDF fields seems to be another possible area to investigate. The integration of the proposed algorithm with some RDF query engine is indicated too.

Acknowledgments

The authors would like to thank the support given by the following Brazilian agencies: State of Minas Gerais Research Foundation – FAPEMIG (Pronem 04611/10, PPM CEX 497/13); Coordination for the Improvement of Higher Level Personnel – CAPES; National Council of Scientific and Technological Development – CNPq (Grants

30506/2010-2, 312276/2013-3, 306694/2013-1); and by the IT Center (NTI) of UFOP.

We also thank the anonymous reviewers for their valuable criticism, which have greatly contributed to the final version of the manuscript.

References

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

Anderson, M. J., & Braak, C. J. F. T. (2003). Permutation tests for multi-factorial analysis of variance. *Journal of Statistical Computation and Simulation, 73*, 85–113. http://dx.doi.org/10.1080/00949650215733.

Barbati, M., Bruno, G., & Genovese, A. (2012). Applications of agent-based models for optimization problems: A literature review. *Expert Systems with Applications, 39*, 6020–6028. http://dx.doi.org/10.1016/j.eswa.2011.12.015<http://www.sciencedirect.com/science/article/pii/S0957417411016861>.

Bennett, K. P., Ferris, M. C., & Ioannidis, Y. E. (1991). A genetic algorithm for database query optimization. In *International conference on genetic algorithms* (pp. 400–407).

Breß, S., Schallehn, E., & Geist, I. (2013). Towards optimization of hybrid CPU/GPU query plans in database systems. In *New trends in databases and information systems* (pp. 27–35). Springer.

Brunie, L., & Kosch, H. (1997). Optimizing complex decision support queries for parallel execution. In *Parallel and distributed processing techniques and applications* (pp. 858–867).

Codd, E. F. (1970). A relational model for large shared data banks. *Communications of the ACM, 13*, 377–387. http://dx.doi.org/10.1145/362384.362685.

Dahal, K., Almejalli, K., & Hossain, M. A. (2013). Decision support for coordinated road traffic control actions. *Decision Support Systems, 54*, 962–975. http://dx.doi.org/10.1016/j.dss.2012.10.022<http://www.sciencedirect.com/science/article/pii/S0167923612002771>.

Davis, L. (1985). Job shop scheduling with genetic algorithms. In *Proceedings of the first international conference on genetic algorithms* (pp. 136–140). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.<http://dl.acm.org/citation.cfm?id=645511.657084>.

Dong, H., & Liang, Y. (2007). Genetic algorithms for large join query optimization. In *Proceedings of the ninth annual conference on genetic and evolutionary computation GECCO '07* (pp. 1211–1218). New York, NY, USA: ACM. http://dx.doi.org/10.1145/1276958.1277193<http://doi.acm.org/10.1145/1276958.1277193>.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 26*, 29–41. http://dx.doi.org/10.1109/3477.484436.

Drezewski, R., Obrocki, K., & Siwik, L. (2010). Agent-based co-operative co-evolutionary algorithms for multi-objective portfolio optimization. In A. Brabazon, M. ONeill, & D. Maringer (Eds.), *Natural computing in computational finance*. *Studies in computational intelligence* (Vol. 293, pp. 63–84). Berlin/Heidelberg: Springer<http://dx.doi.org/10.1007/978-3-642-13950-5_5>.

Elmasri, R., & Navathe, S. (2010). *Fundamentals of database systems* (6th ed.). USA: Addison-Wesley Publishing Company.

Enembreck, F., & Barthès, J.-P. A. (2013). A social approach for learning agents. *Expert Systems with Applications, 40*, 1902–1916. http://dx.doi.org/10.1016/j.eswa.2012.10.008<http://www.sciencedirect.com/science/article/pii/S0957417412011220>.

Feizi-Derakhshi, M.-R., Asil, H., & Asil, A. (2010). Proposing a new method for query processing adaption in database. *CoRR, abs/1001.3494*<http://dblp.uni-trier.de/db/journals/corr/corr1001.html#abs-1001-3494>.

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Upper Saddle River, NJ, USA: Prentice Hall Press.

Ghaemi, R., Fard, A., Tabatabaee, H., & Sadeghizadeh, M. (2008). Evolutionary query optimization for heterogeneous distributed database systems. *World Academy of Science, 43*, 43–49.

Glover, F., & Laguna, M. (1997). *Tabu search*. Boston: Kluwer Academic Publishers.

Golshanara, L., Rouhani Rankoohi, S., & Shah-Hosseini, H. (2014). A multi-colony ant algorithm for optimizing join queries in distributed database systems. *Knowledge and Information Systems, 39*, 175–206. http://dx.doi.org/10.1007/s10115-012-0608-4<http://dx.doi.org/10.1007/s10115-012-0608-4>.

Gonçalves, F. A. C. A., Guimarães, F. G., & Souza, M. J. F. (2013). An evolutionary multi-agent system for database query optimization. In *Proceeding of the 15th annual conference on genetic and evolutionary computation conference GECCO '13* (pp. 535–542). New York, NY, USA: ACM. http://dx.doi.org/10.1145/2463372.2465802<http://doi.acm.org/10.1145/2463372.2465802>.

Guttoski, P. B., Sunye, M. S., & Silva, F. (2007). Kruskal's algorithm for query tree optimization. In *Proceedings of the 11th international database engineering and applications symposium IDEAS '07* (pp. 296–302). Washington, DC, USA: IEEE Computer Society. http://dx.doi.org/10.1109/IDEAS.2007.33<http://dx.doi.org/10.1109/IDEAS.2007.33>.

Han, W.-S., Kwak, W., Lee, J., Lohman, G. M., & Markl, V. (2008). Parallelizing query optimization. *Proc. VLDB Endow., 1*, 188–200<http://dl.acm.org/citation.cfm?id=1453856.1453882>.

Hanna, L., & Cagan, J. (2009). Evolutionary multi-agent systems: An adaptive and dynamic approach to optimization. *Journal of Mechanical Design, 131*, 011010.

Heimel, M. (2013). Designing a database system for modern processing architectures. In *Proceedings of the 2013 Sigmod/PODS Ph.D. symposium on PhD symposium SIGMOD'13 PhD Symposium* (pp. 13–18). New York, NY, USA: ACM. http://dx.doi.org/10.1145/2483574.2483577<http://doi.acm.org/10.1145/2483574.2483577>.

Heimel, M., & Markl, V. (2012). A first step towards GPU-assisted query optimization. In *The third international workshop on accelerating data management systems using modern processor and storage architectures, Istanbul, Turkey* (pp. 1–12). Citeseer.

Hogenboom, A., Frasincar, F., & Kaymak, U. (2013). Ant colony optimization for {RDF} chain queries for decision support. *Expert Systems with Applications, 40*, 1555–1563. http://dx.doi.org/10.1016/j.eswa.2012.08.074<http://www.sciencedirect.com/science/article/pii/S0957417412010500>.

Hogenboom, A., Milea, V., Frasincar, F., & Kaymak, U. (2009). Rcq-ga: Rdf chain query optimization using genetic algorithms. In T. Noia & F. Buccafurri (Eds.), *E-commerce and web technologies*. *Lecture notes in computer science* (Vol. 5692, pp. 181–192). Berlin Heidelberg: Springer. http://dx.doi.org/10.1007/978-3-642-03964-5_18<http://dx.doi.org/10.1007/978-3-642-03964-5_18>.

Ibaraki, T., & Kameda, T. (1984). On the optimal nesting order for computing N relational joins. *ACM Transactions on Database Systems, 9*, 482–502. http://dx.doi.org/10.1145/1270.1498.

Ioannidis, Y. E. (1996). Query optimization. *ACM Computing Surveys, 28*, 121–123. http://dx.doi.org/10.1145/234313.234367<http://doi.acm.org/10.1145/234313.234367>.

Ioannidis, Y. E., & Kang, Y. (1990). Randomized algorithms for optimizing large join queries. *SIGMOD Record, 19*, 312–321. http://dx.doi.org/10.1145/93605.98740<http://doi.acm.org/10.1145/93605.98740>.

Ioannidis, Y. E., & Wong, E. (1987). Query optimization by simulated annealing. *SIGMOD Record, 16*, 9–22. http://dx.doi.org/10.1145/38713.38722.

Khosravifar, B., Bentahar, J., Mizouni, R., Otrok, H., Alishahi, M., & Thiran, P. (2013). Agent-based game-theoretic model for collaborative web services: Decision making analysis. *Expert Systems with Applications, 40*, 3207–3219. http://dx.doi.org/10.1016/j.eswa.2012.12.034<http://www.sciencedirect.com/science/article/pii/S0957417412012754>.

Klyne, G., & Carroll, J. J. (2014). *Resource description framework (RDF): Concepts and abstract syntax*. W3C Recommendations<http://www.w3.org/TR/rdf-concepts/>.

Krishnamurthy, R., Boral, H., & Zaniolo, C. (1986). Optimization of nonrecursive queries. In *Proceedings of the 12th international conference on very large data bases VLDB '86* (pp. 128–137). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.<http://dl.acm.org/citation.cfm?id=645913.671481>.

Lange, A. (2010). *Uma Avaliação de Algoritmos não Exaustivos para a Otimização de Junções* (Master's thesis). Universidade Federal do Paraná, UFPR.

Lee, C., sheng Shih, C., & huei Chen, Y. (2001). Optimizing large join queries using a graph-based approach. *IEEE Transactions on Knowledge and Data Engineering, 13*, 298–315. http://dx.doi.org/10.1109/69.917567.

Li, J., Ding, C., & Liu, W. (2014). Adaptive learning algorithm of self-organizing teams. *Expert Systems with Applications, 41*, 2630–2637. http://dx.doi.org/10.1016/j.eswa.2013.11.008<http://www.sciencedirect.com/science/article/pii/S0957417413009196>.

Matysiak, M. (1995). Efficient optimization of large join queries using tabu search. *Information Sciences – Informatics and Computer Science, 83*, 77–88. http://dx.doi.org/10.1016/0020-0255(94)00094-R<http://dx.doi.org/10.1016/0020-0255(94)00094-R>.

Prud'hommeaux, E., & Seaborne, A. (2008). Sparql query language for rdf. Latest version available as <http://www.w3.org/TR/rdf-sparql-query/>. URL: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.

Rho, S., & March, S. (1997). Optimizing distributed join queries: A genetic algorithm approach. *Annals of Operations Research, 71*, 199–228. http://dx.doi.org/10.1023/A:1018967414664<http://dx.doi.org/10.1023/A3A1018967414664>.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & 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* (pp. 23–34). New York, NY, USA: ACM. http://dx.doi.org/10.1145/582095.582099<http://doi.acm.org/10.1145/582095.582099>.

Sevinç, E., & Coşar, A. (2011). An evolutionary genetic algorithm for optimization of distributed database queries. *Computer Journal, 54*, 717–725. http://dx.doi.org/10.1093/comjnl/bxp130<http://dx.doi.org/10.1093/comjnl/bxp130>.

Shapiro, L. D., Maier, D., Benninghoff, P., Billings, K., Fan, Y., Hatwal, K., et al. (2001). Exploiting upper and lower bounds in top-down query optimization. In *Proceedings of the international database engineering & applications symposium IDEAS '01* (pp. 20–33). Washington, DC, USA: IEEE Computer Society<http://dl.acm.org/citation.cfm?id=646290.686937>.

Steinbrunn, M., Moerkotte, G., & Kemper, A. (1997). Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal, 6*, 191–208. http://dx.doi.org/10.1007/s007780050040<http://dx.doi.org/10.1007/s007780050040>.

Stuckenschmidt, H., Vdovjak, R., Broekstra, J., & Houben, G. (2005). Towards distributed processing of rdf path queries. *International Journal of Web Engineering and Technology, 2*, 207–230. http://dx.doi.org/10.1504/IJWET.2005.008484<http://dx.doi.org/10.1504/IJWET.2005.008484>.

Swami, A. (1989). 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* (pp. 367–376). New York, NY, USA: . http://dx.doi.org/10.1145/67544.66961<http://doi.acm.org/10.1145/67544.66961>.

Swami, A., & Iyer, B. (1993). A polynomial time algorithm for optimizing join queries. In *Proceedings. Ninth international conference on data engineering, 1993* (pp. 345–354). http://dx.doi.org/10.1109/ICDE.1993.344047.

Swami, A., & Gupta, A. (1988). Optimization of large join queries. *SIGMOD Record, 17*, 8–17. http://dx.doi.org/10.1145/971701.50203<http://doi.acm.org/10.1145/971701.50203>.

Tao, F., Laili, Y. J., Zhang, L., Zhang, Z. H., & Nee, A. C. (2014). Qmaea: A quantum multi-agent evolutionary algorithm for multi-objective combinatorial optimization. *Simulation, 90*, 182–204. http://dx.doi.org/10.1177/0037549713-485894<http://sim.sagepub.com/content/90/2/182.abstract>. arXiv: <http://sim.sagepub.com/content/90/2/182.full.pdf+html>.

Tewari, P. (2013). Query optimization strategies in distributed databases. *International Journal of Advances in Engineering Sciences, 3*, 23–29.

't Hoen, P. J., & Jong, E. D. (2004). Evolutionary multi-agent systems. In X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tino, A. Kabán, & H.-P. Schwefel (Eds.), *Parallel problem solving from nature – PPSN VIII. Lecture notes in computer science* (Vol. 3242, pp. 872–881). Berlin, Heidelberg: Springer<http://dx.doi.org/10.1007/978-3-540-30217-9_88>.

Vance, B., & Maier, D. (1996). Rapid bushy join-order optimization with cartesian products. *SIGMOD Record, 25*, 35–46. http://dx.doi.org/10.1145/235968.-233317<http://doi.acm.org/10.1145/235968.233317>.

Van Moffaert, K., Brys, T., Chandra, A., Esterle, L., Lewis, P. R., & Nowé, A. (2014). A novel adaptive weight selection algorithm for multi-objective multi-agent reinforcement learning. In *Proceedings of the 2014 IEEE world congress on computational intelligence*.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. John Wiley & Sons <http://books.google.com.br/books?id=X3ZQ7yeDn2IC>.

Zafarani, E., Derakhshi, M. F., Asil, H., & Asil, A. (2010). Presenting a new method for optimizing join queries processing in heterogeneous distributed databases. In *International workshop on knowledge discovery and data mining* (pp. 379–382). doi: <http://doi.ieeecomputersociety.org/10.1109/WKDD.2010.122>.